Simulation for Improvement of Dynamic Path Planning in Autonomous Search and Rescue Robots

by

Michael Douglas Hasler, B.E. Hons

A thesis
submitted to the University of Canterbury
in partial fulfilment of the
requirements for the degree of
Master of Engineering
in Electrical & Computer Engineering.

University of Canterbury
2009

Abstract

To hasten the process of saving lives after disasters in urban areas, autonomous robots are being looked to for providing mapping, hazard identification and casualty location. These robots need to maximise time in the field without having to recharge and without reducing productivity. This project aims to improve autonomous robot navigation through allowing comparison of algorithms with various weightings, in conjunction with the ability to vary physical parameters of the robot and other factors such as error thresholds/limits.

The lack of *a priori* terrain data in disaster sites, means that robots have to dynamically create a representation of the terrain from received sensor range-data in order to path plan. To reduce the resources used, the affect of input data on the terrain model is analysed such that some points may be culled. The issues of identifying hazards within these models are considered with respect to the effect on safe navigation.

A modular open-source platform has been created which allows the automated running of experimental trials in conjunction with the implementation and use of other input types, node networks, or algorithms. Varying the terrains, obstacles, initial positions and goals, which a virtual robot is tasked with navigating means that the design, and hence performance, are not tailored to individual situations. Additionally, this demonstrates the variability of scenarios possible. This combination of features allows one to identify the effects of different design decisions, while the use of a game-like graphical interface allows users to readily view and comprehend the scenarios the robot encounters and the paths produced to traverse these environments. The initially planned focus of experimentation lay in testing different algorithms and various weightings, however this was expanded to include different implementations and factors of the input collection, terrain

modelling and robot movement. Across a variety of terrain scenarios, the resultant paths and status upon trial completion were analysed and displayed to allow observations to be made.

It was found that the path planning algorithms are of less import than initially believed, with other facets of the robotic system having equally significant roles in producing quality paths through a hazardous environment. For fixed view robots, like the choice used in this simulator, it was found that there were issues of incompatibility with A* based algorithms, as the algorithm's expected knowledge of the areas in all directions regardless of present orientation, and hence they did not perform as they are intended. It is suggested that the behaviour of such algorithms be modified if they are to be used with fixed view systems, in order to gather sufficient data from the surroundings to operate correctly and find paths in difficult terrains.

A simulation tool such as this, enables the process of design and testing to be completed with greater ease, and if one can restrain the number of parameters varied, then also with more haste. These benefits will make this simulation tool a valuable addition to the field of USAR research.

Acknowledgements

Many thanks to Paul Gaynor for providing academic oversight, with guidance in the writing process and encouraging submissions to conferences, as well as help and suggesting ideas when the stress and low-level details overwhelmed the ability to consider the big picture and simpler solutions.

Thanks also go to Professor Dale Carnegie of Victoria University for his help and guidance. His work with robotics and branching into the field of Urban Search and Rescue, being the starting point for the Honours project which preceded this and led subsequently on to this project, to create a tool to aid his research and the research of others.

Additionally the Author would like to extend thanks to postgraduate colleagues and friends, who were there to listen to or discuss frustrating bugs in the code or the search for mathematical techniques and solutions.

Contents

List of Figures

List of Algorithms

List of Tables

Glossary

A

A* A heuristic algorithm, calculating path values based on the distance necessary to traverse to reach a point and proximity of the point to the destination., p. 31.

a priori In advance of a start time. Before examination., p. i.

C

CGAL The Computational Geometry Algorithms Library. A library of mathematical functions and structures, helpful in producing tessellations from sets of points., p. 51.

circumcircle The circle created through joining three points such that they all lie on its edge., p. 25.

circumsphere The sphere which links four points in 3D, such they all lie on its surface., p. 25.

CRASAR Center for Robot-Assisted Search and Rescue., p. 6.

D

D* A heuristic algorithm derived from A*, focused on working backwards from the destination to the start position., p. 31.

D* Lite An enhancement of the D* algorithm, aimed to use a re-use approach much like the LPA* modification of the A* algorithm., p. 33.

Delaunay tessellation A method of generating a set of non-overlapping triangles from a given set of points. The set of triangles being the dual of the voronoi graph of the points., p. 25.

L

LIDAR Light Detection and Ranging. A sensor/technique for gathering precise relative measurements., p. 23.

LPA* A re-use modification of the A* algorithm, limiting re-calculation of path values over successive iterations through a heuristic basis., p. 32.

M

Marsupial Robot teams which involve a larger unit carrying or housing one or more smaller robots, in order to enable transportation across difficult terrain., p. 13.

MeSci The National Museum of Emerging Science and Innovation., p. 17.

N

NIST National Institute of Standards and Technology., p. 17.

P

Polymorphic Robots which are capable of re-configuring modules so that the form factor of the robot is altered., p. 11.

S

SLAM Simultaneous localisation and mapping., p. 28.

Swarms Groups of interacting robots which approach tasks from a co-operative stance., p. 14.

U

USAR Urban Search and Rescue., p. 2.

V

Voronoi graph Voronoi graphs are comprised of cells formed around the points within a set, so that all edges in the graph are of equal distance to the nearest two points and hence any position inside a cell is closer to the encapsulated point than any other point in the set., p. 25.

Chapter 1

Introduction

The first few hours following a disaster are the most critical, with the chance of saving lives being at its greatest. However, in the case of collapsed infrastructure and buildings, the first few hours are also the most dangerous to send rescuers into the site of the disaster. Hence, the concept of using unmanned exploratory vehicles suited to this kind of terrain is being embraced, to map the area and locate individuals trapped in the rubble. Then once it is deemed safe for rescuers to enter the area, they can get straight to saving people instead of wasting crucial time searching.

Many research groups across the globe are now developing task forces of robots which can be used to help in such Search and Rescue operations. This project aims to provide a useful tool for such groups in establishing which algorithms and parameters provide the most appropriate autonomous path planning for their specific purposes or scenarios before a physical robot has been built and without need of a disaster site or mock up, on which to test. This will be achieved through the design and implementation of a software based system to allow a virtual robot to autonomously navigate across a disaster site simulation with the choice from a variety of path planning algorithms and a multitude of variables such as robot dimensions, traversable terrain ascent angles, and data structures. The data structures referred to here include, different types of nodes for use by path planning algorithms, various types of terrain models, restricted and flexible

potential path networks, and also a variety of data storage methods for current and past terrain knowledge.

In order to identify the optimal path through the disaster environment, the robot needs to be able to identify features of the landscape which could be hazardous or insurmountable and determine the shortest path to the destination given the gathered data. As the robot travels and receives new data it will have to dynamically re-evaluate the previously determined hazards and path choice.

Producing a modular open-source design, allows development to be furthered by any interested parties, to reflect design choices they wish to follow or constraints which they have imposed. The addition of new modules greatly expands the potential combinations and scenarios which can be investigated without requiring the replication of the work necessary to produce a platform on which these combinations can be possible.

The motivation which brought about this project's inception, was the creation of a task force of robotic units at Victoria University for the purpose of Urban Search and Rescue (USAR), under the supervision of Dale Carnegie and with the cooperation of other Universities within New Zealand. As such, the selection of features currently present and the basis for decisions such as default values and input method have been guided by that larger project.

To ascertain that the Simulator is both easy to use and can produce usable results, the Simulator tests an individual robot interacting with a varying terrain containing multiple additional obstacles. Values used for the dimensions of this robot and its ability to handle different terrain gradients, are based on a moderately sized four wheel low-profile robot. The focus during testing was on the navigation (and hence also the mapping) of the robot when placed in different locations and tasked with reaching specific goals, with additional exploration not being encouraged more than was necessary to reach the goal.

The original non-graphical prototype from which this simulator progressed, was presented at the 2006 Electronics New Zealand Conference (ENZCon '06), exploring the potential of a USAR simulator, and demonstrating basic behaviour and usage. From the work completed during this thesis, a paper was generated

outlining the design considerations, hindrances and process of creating and implementing a novel simulation platform such as this one. This paper was accepted by and subsequently presented at the 7th IEEE International Conference on Control & Automation (ICCA '09), alongside other papers on the design, simulation and interfacing of other autonomous vehicles and robots.

Chapter 2

Background

2.1 Urban Search and Rescue

Research into the use of robotic units in Urban Search and Rescue began in Japan after a very strong earthquake hit the regions of Kobe and Osaka on the 17th of January 1995 [1]. In the U.S. it was after the Oklahoma Bombings on the 19th of April the same year, that research first began [1]. The 9/11 terrorist attacks were the first major disaster where teleoperated robots were used to try to aid Search and Rescue teams [1]. This globally watched disaster also gave more impetus and research funding, while greatly expanding the number of institutes involved in developing robotics for post-disaster humanitarian efforts. International competitions have also been established to encourage innovation and provide comparison of design and effectiveness.

This field of research has seen numerous different design paths taken, each design choice with its own benefits and flaws. There are various means of robot locomotion, methods of overcoming or avoiding obstacles and techniques or sensors for detecting human casualties. Due to the hazardous nature of the environment, being robust yet cheap has been a primary objective, with self-righting mechanisms and stair/ledge climbing designs also being tried in attempts to minimise the limitations of movement caused by different hazards or insurmountable obstacles.

Tethered teleoperated robots have been the primary choice for USAR robots as using a human agent is considered more reliable to make decisions and because the tether provides the reliable high-bandwidth necessary for live video which a human requires[2]. Autonomous robots can dispense with transmitting a videofeed and as such have more freedom with which to move as they have no tether to get caught and are not limited by the tether length. A tether also provides additional strain/resistance as it must be pulled along, an added means of disorientation (if tension is applied or released to the tether when the robot turns) and a tether could pose a danger as it may trip a human rescuer or if being pulled against/around rumble it may cause a shift or rock fall. One benefit of tethers, is that if robots delve into crevasses and fall or get stuck they can potentially be pulled out via the tether acting as a safety line.

USAR robots can provide replacements for humans and canines in dangerous situations, such as when after-shocks or secondary affects may cause further collapse, or when fires or toxic gases are present, thus avoiding further loss or injury. However, their usage and capabilities in this capacity are far from perfected and this is not the only task where they could provide benefit. Due to being a developing and unproven field, the humans involved with search and rescue are often wary of the presence of robots and trusting the information they may produce [3]. Robots can be viewed by rescue teams as a nuisance/distraction and another obstacle to deal with and which may provide a hazard, in an already dangerous environment. In spite of a distrust of the robots in their conceptual role, Search and Rescue members have found alternative purposes for them such as lowering tethered robots down holes to act as a remote camera/sensor to access structural integrity of lower levels and likelihood of whether they may collapse further.

Alternative ways in which USAR robots can provide assistance to rescuers have been taken into account, with the Center for Robot-Assisted Search and Rescue (CRASAR) developing the use of robots as a delivery system for fluids, medical supplies and sensors, in addition to two-way communication.

One finding of the application of USAR robots in real disaster scenarios has been that designs driven by competition results have been lacking in many ways,

such that they are not suited to real USAR due to the robots not being robust enough and the path planning etc. not being compatible with robots which were[3]. The sensors and detection systems of USAR robots need to be small and robust, however the useful ones are often too large and complex, with the potential to be damaged in a realistic scenario.

2.2 Robot types

There are four common types of locomotion used in ground based exploration. These are wheeled, tracked, walking and snake-like units, which move via undulation of multiple jointed segments. The hybridisation of the jointed segment design with one of the other means of locomotion is referred to as serpentine and an example of such a robot is shown in Figure 2.1 [4]. Wheeled and tracked robots are well suited to autonomous operation with the likelihood of tipping over being much less than walking units and the control being much easier than with walking and snake-like or serpentine robots. However, the much greater freedom of movement of a snake-like and serpentine does allow for very adverse terrain to be traversed, with Figure 2.1 showing one such robot overcoming a piece of terrain no other design would have. Aerial robots can also be used in terrain mapping and have a variety of mobility methods which mimic planes, gliders, helicopters and dirigibles. In disaster scenarios, land based units tend to face a greater range of problems than aerial ones, including shifting terrain, hazardous drops, non-traversable slopes and limited fields of view. However, they can provide greater levels of information due to their closer proximity, especially with regards to location of human casualties.

The manner in which a robot moves and the amount of clearance necessary to do so, and what situations are hazardous or insurmountable, is dependant on the choice of locomotion and hence different robot types may be simulated differently or require different values for some parameters. The intended behaviour of a robot, such as working individually versus co-operatively, or exploring a region versus reaching a specific destination, also affects the design of a simulator and

Figure 2.1: The OmniTread, an example of an articulated serpentine robot design [4].

the choice of features present within it.

Of these traversal methods available to ground based units, there are a few specifics which differentiate them from robots used in other scenarios such as office buildings, factories or entertainment/toys. Unlike the standard arrangement in most wheeled vehicles, wheeled robots for USAR purposes usually have fixed position wheels which are differentially driven, rather than having a pair of wheels which change orientation/angle to steer the vehicle.

All fully tracked vehicles are differentially driven as the tracks do not allow for the wheels to alter direction however the USAR robots have a few track designs which are not common to vehicles. Track designs can be simple, such as the flat layout of a Bulldozer labelled A in Figure 2.2, or may account for desirable behaviour as the trapezium layout of tracks on a Tank does, through allowing for bi-directional traversable over top of obstacles. For some purposes the fourth set of wheels for bi-directional obstacle climbing may be removed to produce an obtuse triangular layout, given as layout B of Figure 2.2, as unidirectional climbing capabilities may be sufficient. In addition to these relatively basic track layouts, another design which is possible for robots is having four sets of tracks, as depicted by designs C and D of Figure 2.2, with two being used for locomotion

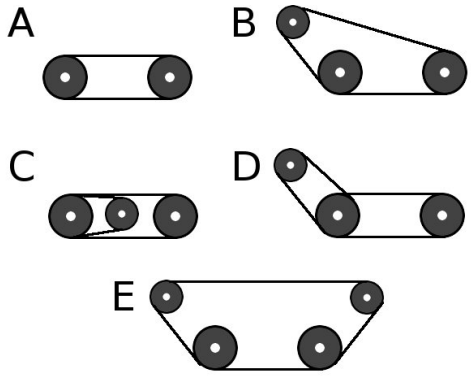

Figure 2.2: A variety of tracked robot designs. A - Basic tracked setup. B - Fixed form climber. C - Flipper arms climber in low profile shape. D - Flipper arms climber in climb configuration. E - Tank like tracks capable of climbing in either direction of movement.

and the others being pivoting arms which can be used to raise the front or rear higher as well as for stair climbing like the obtuse triangle approach. Additionally these flipper arms can be used to right the robot if it rolls or falls, and find itself upside down. Another potential though seeming unutilised approach would be to provide higher point of view for sensors/cameras when the front is raised, to allow them to view past obstacles.

Robots which have legs can either attempt to mimic humans with bipedal motion or adopt a more stable multilegged design akin to an insect or spider, though unlike animals they do not necessarily follow a symmetrical design and can have an odd number of legs. Figures 2.3 and 2.4 show examples of robots designed for bipedal and multilegged motion, respectively. While having more legs provides more points of contact and hence greater stability, each additional leg is another item which must be controlled though often only in terms of extension and placement as the unit will follow an overall guiding pattern of when to move each one in order to move in the desired direction. Designs which have the legs attached radially rather than in two parallel sets, can be omnidirectional as they have no clear front however often the placement of cameras and sensors may dictate a preferential direction.

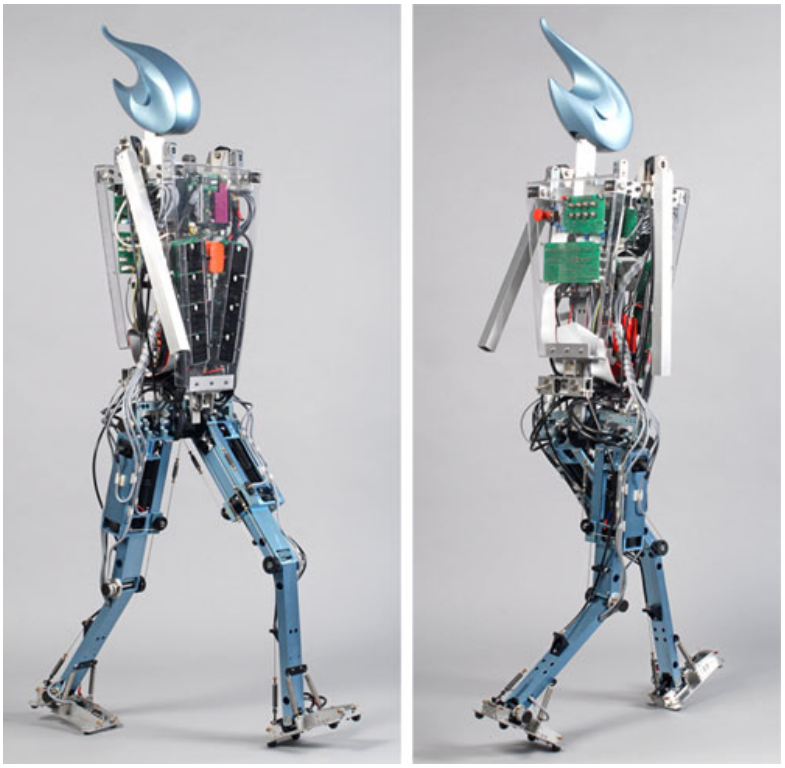

Figure 2.3: A bipedal research robot used to study human motion, viewed from two angles [5].

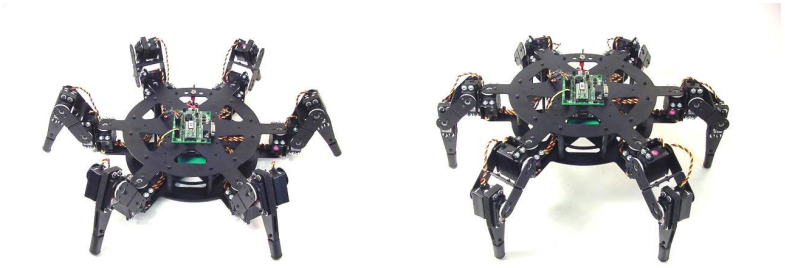

Figure 2.4: A multilegged robot, as commercially available, shown in two positions [6].

The choice of locomotion design effects the navigation in terms of potential obstacles and paths. A common point amongst multilegged walkers, tracked robots, differential-drive wheeled robots and potentially some bipedal robots, is that they are capable of changing direction on the spot. This factor is beneficial for both path planning and simulation as there is no turning circle to account for or need for manoeuvres such as 3-point turns. For serpentine robots, wheeled robots without differential drive, and most bipedal designs, small adjustments in direction are perfectly within their capabilities however drastic changes in orientation can require more room than may be available or manoeuvres like 3 point turns.

Approaches which attempt to climb over obstacles affect the process of hazard identification as they introduce the potential for walls of limited height to not be obstacles. However these stair climber mechanisms hold no substantial benefit with traversing steep slopes other than providing recourse in the event the robot rolls on attempting such a slope.

To deal with the complexities of disaster sites, some research groups have opted to redesigned the whole robotic unit rather than including novel functionality, such as stair climbing, through the addition of modules to standard robot designs. The results of these design branches include robots which can transform into different configurations or contort and shape-shift, so that they can overcome obstacles or fit through tight spaces. Other approaches have seen teams of split purpose robots which aim to specialise in different tasks associated with USAR.

2.2.1 Shape shifters

Taking solutions to a new level are shape shifting robots, which rather than having mechanisms or modules added to a design to overcome certain obstacle types, have the whole unit designed around the idea of crossing the most difficult terrain. Polymorphic robots initially tended to be of a serpentine or centipede-like design as shown in Figure 2.5, which gave them a high degree of freedom, but also meant control was difficult and movement often consumed a lot of power [7]. Balancing efficiency and ease of control against space required to change shape, the next evolution in design were units such as the Amoeba-II, which is shown in Figure

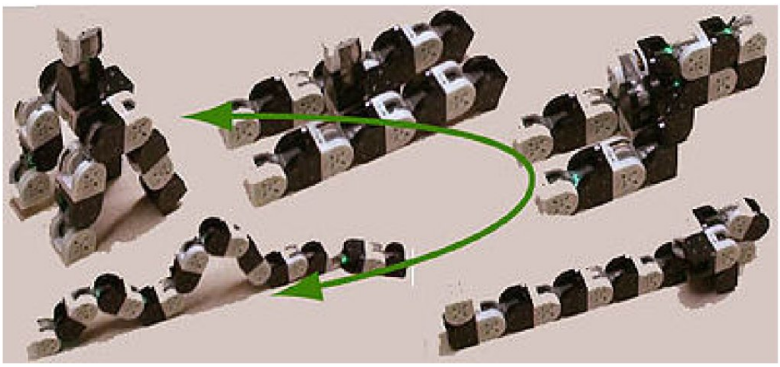

Figure 2.5: The M-TRAN III, a serpentine shape shifting robot, in a variety of permutations [9].

2.6, that can be likened to articulated truck-trailer pairs [7]. These articulated pairs provide a level of compromise, giving the ability for some shape changing but not so much as to be too complex to control, while maintaining much of the efficiency of wheeled or tracked robots. One issue which arises with these simple shape changers is that the space required for them to transform from one mode to another can be much greater. Complex serpentine robots have enough degrees of freedom that the right combination of moves can readjust it in almost any situation. However, for basic designs with few joints, the turning circle/angles required for transformations could make it difficult to find appropriate areas in which to do so. This is especially difficult if the robot is to be autonomous.

While having various configurations in which the robot can traverse across terrain increases the variety of situations which can be handled, these advances in design provide an additional layer of complexity to the already problematic task of autonomous navigation. What the design and focus of current shape shifter research comes down to is the balance of versatility versus power consumption and ease of control for teleoperated robots, with the combination of autonomy and transforming designs being much further off [8].

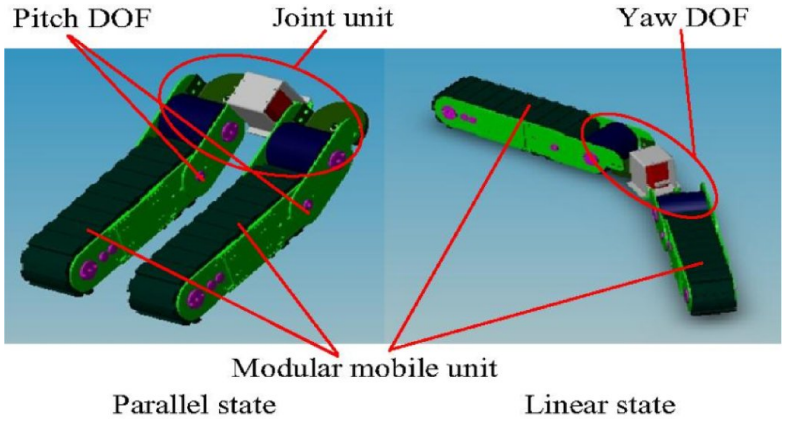

Figure 2.6: An example of an articulated-pair shape shifting robot design. The Amoeba-II shown in two states [7].

2.2.2 Marsupial teams

'Marsupial' robots involve features and abilities being broken down across a set of robots of varying size and shape, with larger ones generally transporting smaller ones [8]. With the separation of tasks, functionality and cost can both be improved, as smaller robots with sensors for finding human life signs can get into tighter spots while safely covering great distances to get to these spots thanks to a larger more robust unit acting as a carrier. To cover a vast area, a swarm of these smaller units can be taken deep into the site and then allowed to radiate out from the carrier, or alternatively be dropped while crossing terrain, to spread out orthogonally from the carrier's path. Thanks to the shelter of the carrier, these smaller units need not be particularly robust and as such can be much cheaper, to the point of even being financially disposable, with losses due to accidents and hazards being tolerable[10].

Unlike polymorphic designs which increase the possibilities and hence complexity of autonomous path planning, marsupial teams are complimentary with autonomous navigation. Each class of robot may have different parameters reflecting their size, hardware and purpose but can use the same approach to navigation. Smaller robots delivered by a large carrier may move in smaller increments and

hence cover a very restricted region, as well as having lower quality sensors which can not detect points as far away. Operating in a smaller region the robots may use a higher resolution/lower scale model, however as they are likely to possess limited memory and processing power, then only relatively simple algorithms can be employed.

2.2.3 Robot swarms

One concept which is reliant on autonomous navigation is using 'Swarms' of robot clones, where information is passed between each unit to improve the modelling of the terrain and also ensure greater coverage of the area without a lot of overlap[2]. Due to the nature of having so many robots on a site, having humans controlling each robot is impractical and would require many expensive workstations and trained operators. Autonomous USAR robots provide a practical solution to this problem. With robot swarms the advantages of autonomous navigation are further embraced, using the ability to transfer information and insert it into other robot's models to build more insightful maps and coordinate activities. Swarms also allow for the positions of individual robots to be triangulated with respect to neighbouring robots and hence over an entire swarm and multiple iterations they can calculate fairly accurate relative positions for all, with any error that does occur being uniform/common to all[2]. This method of localisation via neighbours takes advantage of the more accurate ranging sensors to overcome issues of low quality odometry sensors which may be found in small disposable robots and issues of positional slip which odometry sensors are unable to detect.

As already mentioned, there are issues with the use of wireless communication in disaster sites but these are not overly problematic for swarms due to the different nature of what is being transmitted compared to teleoperated robots. Swarms require lower bandwidth as they are sending specific/quality/valuable data rather than streaming video, would be closer to recipient units, have redundant/multiple communication paths and the data does not have to be received in real-time, with robots still able to operate without interaction, hence wireless issues do not become a concern.

This swarm of clones differs from marsupial teams where the smaller units may communicate only with the central 'mother' unit, being analogous to an ad hoc network as opposed to a network with a central access point. The swarm can be efficient in that the nearer units are more likely to benefit from interaction with the robot, however a central point can provide greater resources for processing and analysing data.

2.2.4 Confined space exploration

One of most dangerous and also difficult places for humans to explore are the spaces created within/underneath fallen buildings and other civil structures, thus specific robots for delving into these tight and dangerous spaces are important [11]. In these instances the danger to a rescuer lies in becoming trapped themselves, coming across gas leaks/pockets and the rubble shifting/collapsing on them thus turning them from rescuer into victim/casualty. Sending robots into these situations not only avoids endangering rescuers but also provides an ability to detect dangerous gases that may be found within the rubble as well as offering a wider range of means to detect humans through the gas/air analysis. The same detection systems, such as nondispersive infrared sensors, which can be used to detect deadly gases could also identify concentration of CO_2, a by-product of human respiration. As well as the outlined danger, it is also highly difficult for a human to delve very far into the rubble without laborious effort or the aid of heavy machinery (something one is disinclined to send onto a disaster site until late in the rescue due to the weight of the machinery shifting rubble and potentially causing more trouble than benefit). The tight, rough and winding tunnels one may find in the interconnecting nooks and caverns within rubble pose little obstacle for robots of a serpentine design, which are both slender and have many points of freedom such that they can contort into a variety of shapes[4]. In some instances, such as the rubble remaining in large pieces, low profile tracked or wheeled robots may also be able to explore much of the region between rubble [11]. In these confined spaces, the line of sight is often drastically reduced due to winding tunnels as well as low cavern/tunnel ceilings (the height of which may vary in addition to the vari-

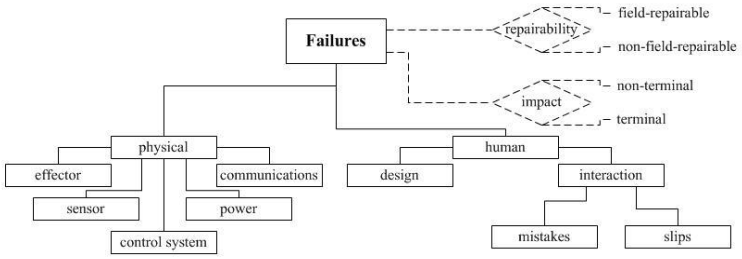

Figure 2.7: Tree diagram categorising areas of failure of mobile robots [12].

ance in the ground upon which the robot is situated). As such, in these regions of occlusion, accurate localisation becomes much more important as the limited line of sight makes it hard to detect errors or disorientation.

2.3 Analysis of Robot failures

Much of the analysis of robot failures focuses on the ruggedness of the physical aspects of the robot and/or its ability to avoid potentially damaging situations [12]. The physical component of the failure tree diagram shown in Figure 2.7, is not overly relevant to this project due to being based in the virtual world. With the human branch of the diagram, which is split between mistakes, slips in control and design flaws, it is mainly the design flaws that are of interest and more so in relation to damage avoidance rather than ruggedness.

2.4 Competitions

In the past much of the focus in locomotive robot design has been on agility and robustness, aimed at exploration vehicles for places such as the moon or Mars where there is a great deal of preparation for a task and numerous people available to work as a team to control a robot. The usefulness of exploratory robots in other applications has been seized upon, as are using them as the basis for developing artificial intelligence. A number of organisations now hold international competi-

tions involving robots [1]. These competitions are aimed at creating more public interest in robotics and allowing comparison between the designs and techniques into which different institutes and research groups have been delving. Some competitions or branches thereof, focus on the human-robot interface and the ability to control the robot in complex tasks, while others focus on the robot's autonomy and cognition.

One of the major competitions being run involving rescue robots, is held by the US based National Institute of Standards and Technology (NIST), which has three different arenas representing disaster site scenarios of differing complexity level [13]. The National Museum of Emerging Science and Innovation (MeSci) of Japan hosts similar competitions to NIST but interestingly, due to the country of origin the materials and scenarios encountered vary quite a lot [14]. The NIST arenas involve typical materials of sky scrapers such as rubble from concrete blocks, tile flooring and carpet. The different architectural styles and cityscapes of Japan mean that sheet rock and wooden building materials can be found in the disaster mock-up along with tatami mats which would be extremely uncommon in North America. Figure 2.8 shows the three of the arenas used in the NIST competitions, while the arenas present in the MeSci competition can be seen in Figure 2.9, with it possible to notice some of the similarities and differences in the materials used for each competition.

For both competitions the three arenas are colour coded as yellow, orange and red in order of increasing complexity. The yellow arenas are large and designed to test the sensors and algorithms used in mapping and planning. Some areas of the arena can be altered in real-time, for example closing doors or adding blockages to passages, to test how the robots react to changes in known regions. Overall the yellow arenas, as shown in Figures 2.8 and 2.9, look like evacuated buildings rather than the sites which have sustained damage from some of a disaster.

With the orange arenas the focus shifts to testing the agility of the robots. Various flooring materials which may affect locomotion or odometry, are used across the arena as Figure 2.8 shows and there are some raised sections such as the one in Figure 2.9 which are added to create a multi-tiered environment. In

addition to ramps, more difficult means of reaching the upper level, such as via stairs or a ladder, are present to encourage and test various climbing techniques. In addition to simple obstacles hindering robots, the orange arena contains hazards which pose a danger to the robots, such as items which if disturbed could cause a minor collapse or holes in the upper level into which a robot could fall.

The most difficult scenarios are present in the red arenas, which as can be see in Figures 2.8 and 2.9, begin to resemble actual disaster environments. Building materials are strewn around creating a terrain which is not only difficult to traverse but also can be problematic for sensors and mapping due to translucent items such as plastic sheeting or discontinuous surfaces like wire meshes, gratings/vents or pallets. In addition to items which if knocked could collapse around or on a robot, the red arena includes flooring which is unstable and may potentially collapse underneath a robot. This type of collapsing hazard provides a significant threat to designs which may attempt to approach USAR through heavy duty robustness and an over abundance of sensors, resulting in heavy robots.

These competitions provide a base platform which allows USAR designs to be benchmarked and compared, however the scenarios are far from realistic. Robots which can survive and negotiate the arenas, will not necessarily be capable of surviving or traversing a real disaster site and the algorithms/automation is in general not portable to other robots which are suited to being used on real disasters.

The most widely known string of competitions have been the ones sponsored by the Defense Advanced Research Projects Agency(DARPA), a US Military research group, with the competitions being the Grand Challenge and the Urban Challenge [15]. The robotic side of the competitions is not as clear to some due to standard vehicles being used, which outwardly do not match public perception of what a robot is. However, these vehicles have been outfitted with many sensors and other robotic controls in order to attempt to autonomously traverse a rugged terrain.

The RoboCup is a group of competitions which are more aligned with public perception of robotics, with units being of a much smaller scale than a vehicle and usually having a much more science fiction like appearance [16]. The first

Figure 2.8: The three arenas of the NIST competition [14].

Figure 2.9: The three arenas of the MeSci competition [14].

competitions involved playing soccer and were split into different size categories, with a humanoid league later being added as well as competitions focussed on USAR.

2.5 Simulators

In terms of 3D Robot Simulators a number of commercial alternatives exist, with a number of very limited open source ones also available [17]. Simulations have in past been limited to a numerical basis (such as those based upon Matlab) with limited or no graphical visualisation, but 3D simulators have over time been adopted by many. In terms of graphics, many have used custom rendering engines which require their own upkeep, rather than widely used pre-existing ones which are likely to stay current with improvements in technology via large development teams or communities. However, there is a trend of moving towards commercial graphics engines or pre-existing open source engines, which has made the uptake of 3D graphical simulations easier as it no longer requires a team/number of developers to implement the graphical side of a simulator.

Most of the simulators are purpose built for certain robots or specific test arenas and in general are not of much use for the wider research community. Instances of simulators used to recreate specific arenas, like the three NIST arenas for example, are analogous to implementing an analogue solution upon digital equipment in that they replicate solutions which had been restricted by limited capabilities and discard all the benefits which the modern technology can provide. Running in the virtual world, a computer simulator allows for drastic alterations to be made to test environments without the time, labour or material costs associated with altering or adapting a physical test arena for each new layout to be tested.

Except for one instance, all the search and rescue simulators encountered have been designed for the purpose of testing teleoperation capabilities and interfaces, focussing on improving the interaction aspect of the failure tree shown in Figure 2.7, with research into and testing of, autonomous navigation being an ignored area of potential application for simulators. At present the only autonomous nav-

igation simulator found by the Author, is the Unified System for Automation and Robot Simulation (USARSim)[18], which was not encountered until the completion of this project. USARSim uses the Unreal Tournament (UT) game engine, which it requires to be purchased separately, and has been the basis for the RoboCupRescue virtual robot competition. While providing a very picturesque high-fidelity simulation for free and purporting to being compatible with Unix-based systems, USARSim appears to only be readily available as a pre-compiled executable file for Windows or as source code developed for Windows, and requires the purchase of the proprietary UT graphics engine. So as at present no other modular or truly open source simulation platforms could be found.

2.6 Autonomous Navigation

The primary constraint of almost every mobile object is energy, be they baseballs, mobile phones, humans, cars, or robots. The object's capacity to store energy, the frequency of events or locations at which more can be acquired and the efficacy of use, all determine how an object behaves. Some robots have automated recharge points which they periodically must find and return to, other robots will have humans replace or recharge battery banks and then there are a few robots such as the Mars Rovers which once unable to obtain more energy are simply left where they lie. In all cases, the paths travelled by the robots and how they are chosen play an important role, from ensuring the robot finds the recharge point to allowing the robot to operate for as long as possible before it is useless.

The environment robots are placed in are often unknown, with data being received from sensors which give relative range data. The lack of *a priori* data means the robot has to dynamically create a representation of the terrain, so finding perfect paths is unlikely. Dead-ends will occur in the terrain, so the robot will at times need to back-track and thus the robot must keep data about the terrain for some time, rather than discarding upon passing.

In order to identify suitable paths through the rubble, a robot needs to be able to identify features of the landscape which could be hazardous or insurmountable

and determine the shortest path to the destination given the known data. As a robot travels and receives new data it will have to dynamically re-evaluate the previously determined hazards and path choices.

2.7 Terrain modelling

One section crucial to both identifying hazards and planning paths is the interpretation of data from the input sensors, which is dependant on the nature of the sensors. Sensors may be omnidirectional or have a limited area of detection. In instances where the received data comes from a forward facing directional sensor, such as some form of range finder, regions can often be occluded by obstacles or rises in closer areas and so when ascending a section of terrain the following descent is likely to be unknown. How the height value of an unknown XY coordinate is interpolated depends on the method of modelling the terrain. Models are generally derived from the point cloud of data received about the real terrain, and may involve turning the points into a non-overlapping mesh representing the surface. For omnidirectional sensors occlusion may also occur, however as a robot moves and new data are received, an omnidirectional sensor may view an occluded region more times and from a greater range of angles, giving it more opportunities to detect the previously occluded region.

2.7.1 LIDAR

One method of determining distances is through using Light Detection and Ranging (LIDAR). While live camera feeds are highly beneficial to humans teleoperating a robot, they are not intrinsically of benefit to an autonomous robot which would need to process the images for edge detection and try to extrapolate relative distances based on the visual expansion or diminishment of points and knowing its own motion. So LIDAR may be used instead to gather sets of relative measurements from which to build a model. LIDAR measures the distance to the first object a laser encounters in a straight line path and knowing the orientation and

inclination of the laser, this measurement can be used to calculate the relative position of the object the laser intersected. The principles behind LIDAR are much the same as with radar systems but with LIDAR operating within the ultraviolet, visible light and infrared parts of the electromagnetic spectrum as opposed to using microwaves or radio waves. The benefit of using shorter wavelengths such as IR is that tiny objects can be detected/imaged, thus differentiating the potential uses for LIDAR from radar, which is better suited to tracking large objects such as cars, planes or ships for which fidelity/resolution is of little importance.

While LIDAR is good for monitoring the proximity of obstacles, due to measurements being relative to the robot, the building up of a terrain map can suffer problems if the robot's motion is imprecisely known, it slips or is disorientated. The potential for errors can be dealt with through coupling LIDAR with absolute positioning systems such as GPS or determining a localised position through model matching, the proximity with other robots or active beacons.

The basic process by which distances are determined with LIDAR is through pulsing a laser and then judging the return flight time based on the phase difference between transmission and receipt, and knowing the speed of the signal. There is a limited range within which measurements can be made due to the signal strength weakening and limitations of detection.

2.7.2 Tessellation

To create a terrain mesh and hence a model which can be interacted with to determine surfaces, their gradients and collisions with them, the point cloud of input data has to be transformed into a set of non-overlapping shapes. This process is referred to as tessellation and can produce sets of polygons of various sizes and shapes, or sets of specific shapes such as triangles, which is also known as triangulation. As all polygons can be broken down into submeshes of triangles, both paths of modelling are closely related. It is preferred to use triangles for rendering large interactive terrains due to the simplicity provided through the uniform number of vertices and neighbours. There are numerous ways to join any given set of points and hence there are many methods of tessellation, however one of the most

widely used and thoroughly tested is the technique put forward by Boris Delaunay in 1934[19]. Delaunay's approach has had much mathematical scrutiny and has been found to produce quality meshes which are excellent for the purposes of graphical modelling.

Delaunay

Delaunay tessellation is a well established method of turning any arbitrary model in two or three dimensions into a set of triangles or tetrahedra, respectively. The act of transforming a set of points/vertices into a set of non-overlapping triangles or non-intersecting tetrahedra which fully embody the model is certainly not a simple one. A vast number of possible combinations/permutations exist and as such Delaunay tessellation is based on a number of mathematical properties. A benefit of Delaunay triangulation is that due to the mathematic principles involved, it attempts to maintain optimal internal angles for triangles and tetrahedra which are hence produced in more uniform shape. As such, two dimensional meshes are usually devoid of slivery triangles, however in three dimensions slivery tetrahedra may still occur dependant on the terrain. At the core of the Delaunay tessellation is the feature that the defined set of edges is the dual of the Voronoi graph for the input points. A Voronoi graph being a set of non-overlapping cells, each position around a point such that any position inside the cell is closer to the enclosed point than any other point within the set. Following on from this property, no point may lie within the circumcircle or circumsphere of a triangle or tetrahedra. This characteristic makes the incremental addition of new points much simpler, as it provides a test for determining which triangles/tetrahedra are affected by the new point's insertion and thus identifies which set of points needs to be re-processed to maintain a valid Delaunay tessellation. A number of algorithms exist to build Delaunay tessellations in two or three dimensions, with the principle applying to higher orders as well but with the complexity of ascertaining the validity of tessellation and hence the complexity of creation greatly rising with the inclusion of each additional dimension. A popular method of constructing Delaunay tessellations is the Bowyer-Watson algorithm, which starts with a large all encompass-

ing triangle/tetrahedra that tessellates as points are incrementally inserted within it[20]. There are two criteria which must be maintained for validity, firstly that no degenerative cases exist whereby a loss of precision leads to ambiguity with respect to whether additional points exist on a circumcircle/sphere. The other criteria requires that the polygon/polyhedron created by the set of affected points must be 'point convex', whereby the points/edge form a convex hull. A convex hull being when the hull or boundary enclosing a set of points is convex and has no regions within it not being covered by triangles.

A modification to Delaunay tessellation is when a set of edges are given which must not be broken or removed. Because of the restrictions imposed by this, this is referred to as a constrained Delaunay tessellation. Constrained tessellation further increases the complexity of tessellation and as such is currently only possible with two-dimensions, as an equivalent method for three has not been discovered. The benefit of these constrained tessellations would be having them coupled with edge detection algorithms, which determine the presence of edges from visual input. The lines found by the algorithms would help to ensure a greater accuracy in the depiction of the environment [21].

2.7.3 Data culling

To ensure the quality and accuracy of models, a large quantity of points are required. However, the amount of data required to gain confidence in the accuracy of a model and subsequent alteration to it, can be excessive in terms of data storage. In order to reduce the resources used, selective filtering of points is necessary in judging which to use and keep/insert into the model.

Dependant on the terrain being viewed, the data or groups of points within the data, may be densely grouped when faced with an upwards slope or an obstacle is nearby while a downwards sloping surface or being at the top of a ridge/cliff may result in the points being spread over a greater area. To ensure having enough data about a region in the latter case, the sensor/scan increments need to be small/fine and hence for the former case there is an excess of data, which needs to be culled. Eliminating points based on proximity, in order to produce a more

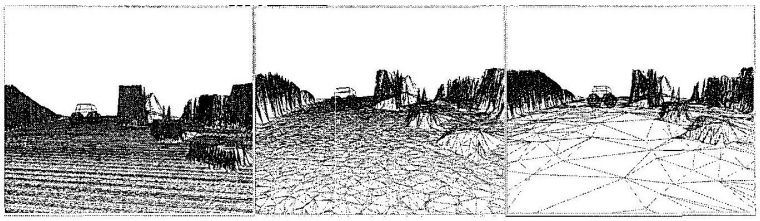

Figure 3: Example of triangulation result. Left: without pre-processing; Center: with proximity suppression; Right: with coplanar point elimination.

Figure 2.10: Comparison of a terrain model against results from applying data culling techniques [22]. Left: without pre-processing; Centre: with proximity suppression; Right: with coplanar point elimination.

desirable level of fidelity/resolution, is one method of culling the data which is very effective. Large flat surfaces may potentially exist within a terrain, with little value being provided through these being tessellated by numerous triangles, so combining neighbouring triangles which exist within the same plane, can help reduce the number of points used within a model. However, establishing with certainty whether a surface is flat and whether modelling it with a few large triangles is accurate, is difficult as it may only appear so due to the presence or combination of, sensor error/accuracy, over zealous data culling or occlusion of points due to angle/position. Sensor errors/noise can also conversely result in a flat surface being modelled by triangles which are not coplanar, so tolerance/precision factors must be involved with the removal of data, which thus provides another aspect of the modelling and culling which may lead to erroneously judging a surface to be flat, or potentially judging a coplanar region to be larger than it is. Figure 2.10 demonstrates the benefits of applying pre-processing data culling methods to a terrain model, with the frames (going from left to right) containing progressively fewer triangles while producing a model of the same terrain. The effect of eliminating points which are coplanar is particularly noticeable for the large flat region in the foreground of the models.

In addition to culling received data so as to minimise the resources allocated/necessary for producing an accurate terrain model, further filtering and analysing already held data to limit the size/expanse of the terrain model, provides further

benefits. Having survived the filtering upon initial reception, data culled at a later stage will have had some value, so attempts to extract/acummulate/concentrate that value into less data may occur, rather than simply discarding/deleting data.

Relaxing the conditions/thresholds of retention of coplanar points, when the points are outside a distance of likely re-encountering/interest such that groups of coplanar faces/triangles are recombined is one such method of decreasing the quantity of data without significant impact on the quality, however dependant on the model or method of tessellation, removal of points may be processor intensive.

2.7.4 Simultaneous localisation and mapping

"The 'solution' of the SLAM problem has been seen as a 'holy grail' for the mobile robotics community as it would provide the means to make a robot truly autonomous [23]."

Simultaneous localisation and mapping (SLAM) is an area of research which attempts to use relative measurements of the terrain to both map the region and determine the robot's position within the terrain. As building an accurate terrain model requires knowing the location/relative position and localisation requires accurate maps, SLAM is somewhat of a 'chicken and the egg' conundrum. The eventual aim is to be capable of handling the 'correspondence problem', which involves being able to identify/recognise a set of points common to two or more views/images of the same object/area. In the case of mobile robots this means identifying previously encountered terrain when approached from a different angle and position, by comparing the held map/model with the current view/data. A simple example of the goal for this field of research is the idea of a robot entering a previously visited room via an alternate door, such that the room and contents are now being viewed from the side rather than front, recognising that it had been there and hence both determining its location and further improving/building upon the terrain map. When applied in two dimensions online scan matching which compares held data with scanned data to identify patterns in order to estimate the robot's position is not too difficult. However, when applied to 3D terrains it becomes much more problematic as the positioning (height) and orientation(incline)

of a robot has to also be accounted for [11].

While this project does not aim to solve the SLAM problem, the style of map building is similar to other autonomous systems such as the type created for this project and highlights the kind of issues which may be faced when mapping a terrain using relative measurements. Maps are built which revolve around the geometrical relevance/consistency (shapes, size, spacing etc.) of the terrain and being as accurate as possible in this respect, as opposed to identifying topological features such as altitudes/contour lines which are more relevant to large scale approximate mapping. Over time, as features are observed from various positions, the accuracy of estimates/modelling increases. Because all measurements within the model are relative, it is not just the re-observed features which improve but also other previously viewed regions, as the affect of the improved estimates propagates throughout the model. This propagation/convergence, means that localised maps provide for high levels of accuracy in terms of relative measurements within a model, however they can lack accuracy with regards to where they believe they, and the terrain, fit on a larger scale, with Figure 2.11 demonstrating this characteristic. Potentially, scanning matching with maps from other robots maybe necessary to reconcile global positioning (as mentioned in swarms/ad-hoc networks). An issue which can be present when using relative positioning to build maps, is that the environment may alter over time and the disparity between the held model and real terrain can flow through to affect the robot's localisation. Using a probabilistic approach allows for variance and subsequently adjustment over time, allowing the model and localisation to eventually converge with the (newly altered) real terrain. For non-SLAM based systems it may be necessary to instead discard all the data and begin again or to have some measure of tracking trust/confidence about regions.

2.8 Path Planning Algorithms

There are multiple types of path planning, which are commonly split into two main categories. The first category is topological planning, which is when the terrain

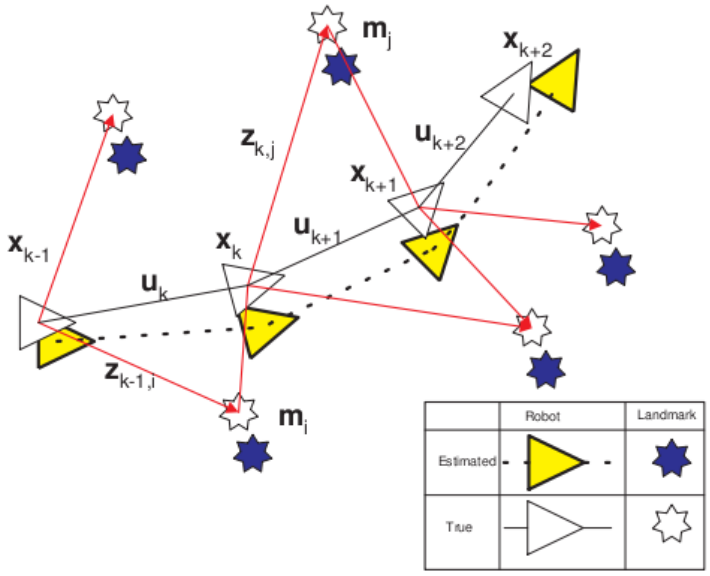

Figure 2.11: Diagram highlighting the accuracy of measurements within a model but variance from real world locations.

to be crossed is known in advance. In most situations this is the preferred style as it produces shorter paths. However, in exploratory applications, topological planning is not possible, so a second category known as Dynamic planning or Sensor Based planning is used. Dynamic planning is where little to no data are known *a priori* and thus this style can result in substantial back-tracking due to encountering dead ends [24, 25, 26, 27].

There are many different algorithms which can be used in path planning to meet many different objectives [28, 29]. Some algorithms are very simplistic and designed to find any route through an environment, while some aim to find the perfect path with no consideration of memory size or processor limitations. There are also those in between which make different concessions in order to fit different limitations or desires. An algorithm which tries every combination of points is called a Blind Search and will find a path but can take a long time to run as, “For hard combinatorial problems the search fringe often grows exponentially with the search depth [28].” Heuristic algorithms use one or more predefined criteria to find

acceptable solutions to a problem, an example being comparing sizes and placing larger items in a box first when trying to pack [30]. A related set of algorithms are some metaheuristic ones which are referred to as Greedy algorithms [31]. At each step of reaching a solution, a Greedy algorithm will choose the locally optimal option in the hope that this will lead to attaining an optimal result. For path planning, metrics are chosen so as to compare or estimate potential paths in order to reduce search time by prioritising the evaluation of paths which are likely to be shorter or more optimal. In general, "Heuristic searches will (usually) find ANY path, but will do so faster (usually) than blind search [29]." This has led to a range of heuristic algorithms being adopted by or developed for autonomous navigation, such as A*, Dijkstra's, D* and D* Lite [28, 29].

2.8.1 A* Algorithm

The A* algorithm is a heuristic approach to path planning which involves calculating the distance travelled within the network in reaching a node (g-value) and the direct line distance to the goal from that node (h-value). These two values are then combined as the node's f-value and used as a basis for prioritising which potential paths are evaluated first. The A* algorithm is best suited to quickly generating reasonably optimal paths in known environments, requiring less processing time than many greedy algorithms. The A* algorithm has the benefit of being able to recover and choose less optimal intermediate steps if a locally optimal choice does not lead to any solution. Due to the nature of the algorithm, any changes in the network require full reprocessing of the planned/potential paths [32, 33].

2.8.2 D* Algorithm

One adaptation of A* is the Focussed Dynamic A* (D*) algorithm which is designed to accommodate changes in the potential path network and as such reduce the amount of recalculation which occurs on subsequent searches [34]. D* can be considered a reuse approach, whereas A* purely involves replanning, thus making D* more suitable for environments in which dynamic variations are likely.

A key difference of the D* approach is that it works in reverse to A*, finding paths from the goal to the current location. This reversal is especially beneficial when alterations occur nearer to the current position than to the goal, as the need to recalculate values is propagated from the point of change back to the current position [34].

2.8.3 LPA* Algorithm

Another approach which looks to modify A*, in order to minimise reprocessing when changes are encountered, is the Lifelong Planning A* (LPA*) algorithm [33]. This speed enhancement is achieved through combining the original A* algorithm with an incremental search algorithm. The particular incremental search used within LPA is the Fixed Point Problem version of the Dynamic Strict Weakly Superior Function (DynamicSWSF-FP) [33]. Much like A* reduces processing through prioritising which paths are more probable to lead to an optimal solution, the incremental search identifies which paths are relevant to the modification such that only they are reprocessed.

The means of identifying which paths should be prioritised for recalculation, is the introduction of a new variable called the rhs-value. The rhs-value is the minimum value produced when the neighbours of a selected node, have their g-value summed with the distance between the node and the neighbour. This value acts as a one-step look-ahead and can thus can be used to determine whether the node's g-value is no longer consistent with its surrounding neighbours. If the rhs is lower than the g-value then the corresponding neighbour becomes the node's predecessor or if the rhs is higher then the g-value of the node is made infinite and the local area needs to be evaluated. Nodes being evaluated get placed on a stack, in order of [Min(g, rhs) + h); min(g, rhs)]. Eventually a path to the goal is found and made consistent, any nodes left in the queue are left with infinite g-values and not recalculated as they are of non-consequence as their f-value is guaranteed to be higher than paths around them [33, 35].

If changes occur close to the position of the robot, then the LPA* algorithm can be worse in terms of processing than the A* algorithm. This scenario requires

extensive updating and re-analysis in conjunction with the propagation of updated values, which is more complex and can be more time consuming than analysing the whole network with reset values.

2.8.4 D* Lite Algorithm

Building upon the LPA algorithm's vision to combine a heuristic algorithm with an incremental search algorithm, the D* Lite algorithm enhances D* so as to further reduce processing time [35]. D* lite is a derived version of the D* algorithm, with the two having a relationship similar to the one between the LPA* and A* algorithms. With D* Lite the program flow is much simpler and the processing involved in path analysis is substantially less than the original D*. One enhancement over LPA* is that the priority queue requires less reprocessing to reorder as values held within are relative, so values added when the robot is in a different position can be equated on equal terms to those older values. An improvement upon D* is that only one tie-breaker is necessary to judge nodes of similar value during comparison [35].

2.8.5 Multi-resolution Field D* Algorithm

As many implementations of the aforementioned heuristic algorithms are based on a uniform grid of nodes, there is the difficult task of balancing the resolution, and hence accuracy, of the model versus the required data storage and processing for a set number of nodes. Without moving to a fully flexible unconstrained network of nodes, some improvement can be made through allowing varying resolution levels to be used in different regions [36]. This modification to D* allows areas of greater danger or interest to be held/modelled with more nodes than regions which have a more homogenous and more easily modelled shape. In addition to producing networks of nodes which provide equivalent quality at reduced resource cost, in some case the paths generated are more optimal. The improvement in the paths lies with there being a greater range of paths/angles which may occur between two points as is shown in Figure 2.12 [36].

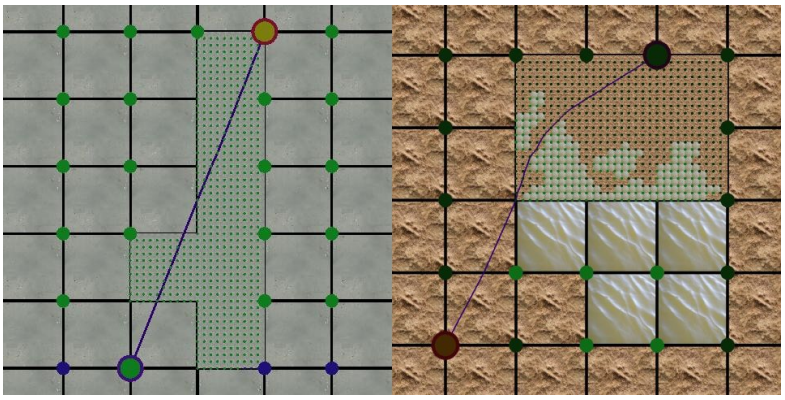

Figure 2.12: Multi-resolution Field D* produces direct, low-cost paths (in blue/-dark gray) through both high-resolution and low-resolution areas [36].

Chapter 3

Simulator Design

With research into USAR robots and the testing of designs often being very time consuming and hard to do in a realistic manner, virtual simulations are necessary. The purpose of this simulator is to provide an open modular platform on which researchers can simulate the response of different robots and path planning system under various scenarios. To be of maximum benefit in understanding the behaviour of a robot in these situations, the simulator provides a number of means through which the movement of the robot can be visually displayed and as such transfer situational awareness to the researchers. The irrlicht graphics engine, which is described further in Section 3.1, not only provides the visual front-end component of this system but also provides realistic input to the robot. The design of the system is tailored towards the use of a single robot tasked with reaching a given position. For instances where the goal is to cover as much area as possible and thus expand the mapped region, this simply requires the path planning algorithm to dynamically choose and adapt/alter the robot's destination. The system would also be capable of simulating multiple robots on the same terrain with some very minor modifications to the code, such as creating more robot objects and converting the current robot pointer held by the simulator to an array of pointers to the multiple instances.

The inter-relation of the objects used within the simulation program is shown in Figure 3.1. The overall control of the simulation lies within the Simulator

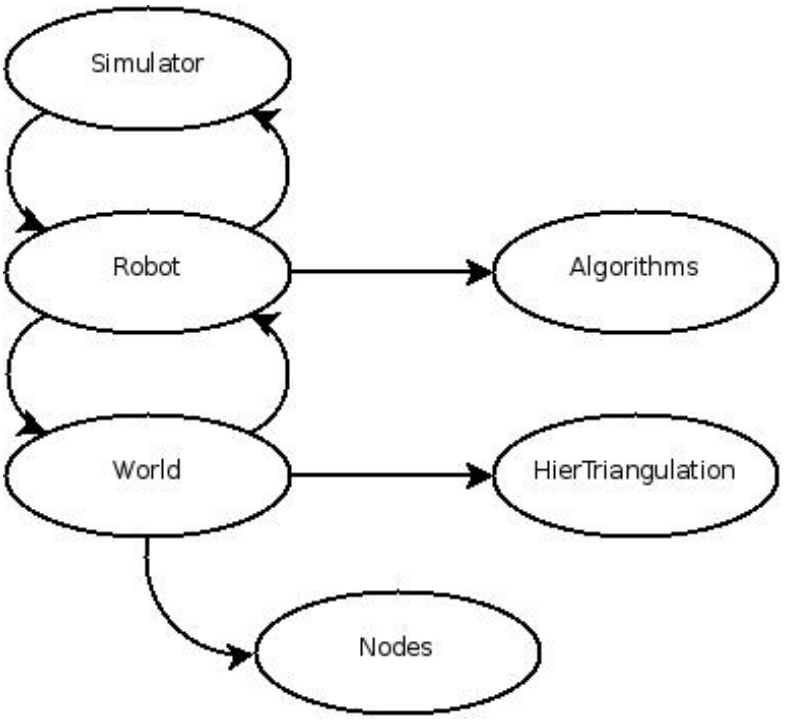

Figure 3.1: A UML depiction of the interconnection between objects in the simulation.

object, which setups up the graphical elements of the simulator and controls the calls to functions within the Robot object. The Robot object is tasked with passing the data generated by the Simulator, to the World object which creates an internal model of the terrain. The process of moving the robot within the simulation is also carried out by the Robot object based on the path planned by Algorithm functions. The terrain model created by the World object uses the CGAL library, and then after analysis of the model, sets of Node objects are created. The purpose of the Node objects are to provide a simplified and reduced network with which the path planning can work to produce potential paths to follow. The full source code for this project can be found in the Appendix, with pieces of pseudo-code being given within this chapter in order to illustrate the nature and operation of functions.

3.1 Simulator

Other simulation undertakings have focussed on the user interface and the replication of specific environments, for testing of tele-operation against standard benchmarks such as NIST USAR Arenas [37]. Human-Robot interfaces have also been of interest with real world tele-operated or semi-autonomous robots, where increasing levels of freedom, sensors and data can lead to cognitive overload for an operator [38]. However, for this project the focus was more on the methods of interaction of the robot with the simulator. As such the simulation environment was mainly an abstract and modular base on to which specifics could be added or altered, with the human interfacing with the robot being superficial, simply providing visual feedback on general navigation such as data relating to hazard identification, path planning and the path implementation/chosen.

In addition to the shortened development time of using a pre-made graphics engine for simulation, the game-like nature of the graphics engine gives familiarity to allow easier understanding of behaviour and also lends itself to later interface interaction/design. The particular graphics engine chosen for this project was Irrlicht [39] as it is open-source, cross-platform and was found to be easy to learn and use. The simulation consists of three camera views, of which one is a static bird's eye view and the other two are dynamic. The dynamic views allow for a better sense of what the robot encounters, with one following the robot's view and the other being available to the user to navigate in order to watch the robot from different locations and perspectives. A greyscale bitmap is loaded by the simulation as a heightmap, from which the graphics engine creates a terrain mesh onto which additional obstacles or items may be added. This terrain mesh takes the place of the real world and functions both as the basis for the graphical representation so a user can see the environment and also as the model of the real environment, with the simulated input being derived from this model. Figure 3.2 is a screenshot of the simulation from basic obstacle avoidance testing and shows the robot, modelled as a car, approaching a section of terrain which is littered with a number of large insurmountable grey objects. A translucent white marker

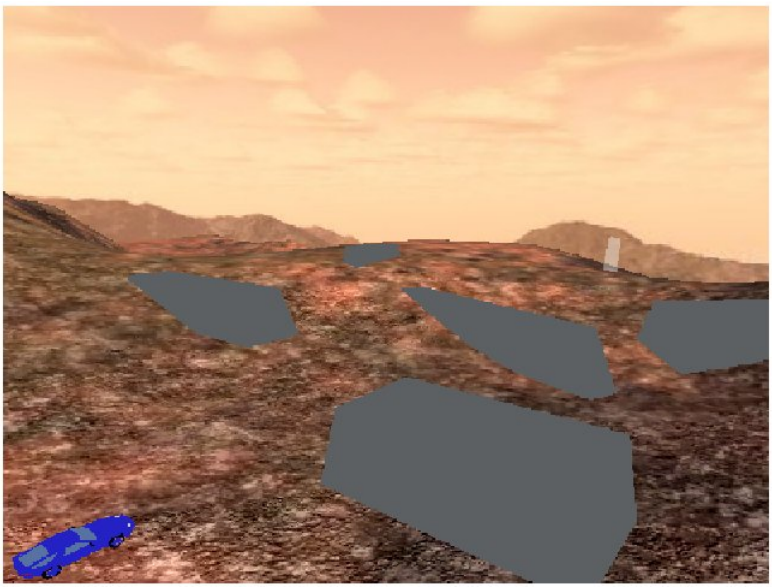

Figure 3.2: Simulator screenshot depicting Robot approaching terrain containing obstacles.

is visible, the purpose of which is to indicate to a user where the robot's goal is located. Due to the resolution of the images and the texture/material applied to the obstacles, it is hard to clearly identify the vertices and edges of the obstacles from Figure 3.2. However, through the robot's perception the position of vertices and edges, and hence the exact shape, can be easily detected if the areas were to be scanned.

3.2 Modelling LIDAR input

The common method to build up a set of LIDAR measurements for a position, is to alter the direction and inclination of the laser in fixed discrete increments using pivoting points and mirrors. The result is that these points are dispersed/spread radially and as such the method through which the simulator scans the terrain and provides data to the robot needs to reflect this. Additionally, if the laser does not intersect a surface within a set range then no measurement can be produced, as

the signal strength upon return would be too weak to detect.

An alternative method for collecting LIDAR measurements involves using a lens to disperse/spread the laser beam and then the data points are taken through sampling discrete points in the sensor behind the lens. Due to the lack of moving parts this provides a more robust input method, suitable for USAR purposes. However, both cases can be modeled in the same way.

The number of input points and their angular dispersal, will depend on the increments by which the laser is adjusted and through which range/arc, or by the lens shape and sensors.

The scanning of the terrain was designed to produce points which emulate the kind of dispersal expected from light passing through a lens, whereby points will be closer spaced in the centre and become less so at the periphery. The range within which data points can be detected has been designed to be variable so it can be set according to any specific sensor, as can the dispersal of points.

Having the horizontal increments being fixed across all vertical increments meant greater spread at edges. The vertical increments increased in sized as they spread out from the sensor in order to cover a greater range but avoiding having to deal with too many data points. This choice of increasing increments as opposed to fixed ones, could be considered the first stage of culling data. This accounts for the affect of vehicle incline due to the variation in gradient of surface at the times when it gathers more data, plus when terrain in front of the robot is sloping up or down significantly. The downward view for close points is particularly important as it is necessary to detect steep drops in front of the robot. It is less important to obtain many points in the upward view as it will usually only produce data when the robot is angled downwards and facing a nearby rise.

An offset to account for the camera's height with respect to the simulated terrain was necessary (since the position coordinates held by robot relate to the point of contact on the terrain surface) to get downward/nearby points for flat or declining planes, otherwise only obstacles will be detected.

The retrieval of scanned input points occurs at regular/periodic intervals, specifically after each movement. The simulation has the robot pause when it gathers the

new data, because this was seen as a safer options because it means the robot does not risk hitting a hazard while it analyses the data and choose the next movement. This stop-start technique may introduce some jitter, however scanning while moving would also be likely to produce jitter as well as the robot rocks and sway while crossing variations in the terrain.

3.3 Terrain Modelling and Data Storage

A key part of navigation is the model used to represent the environment; how is data for it gathered, how it is interpreted, what data points are discarded or kept and when is the decision to remove them made. Many different methods of representing the terrain and storing the data were considered because "If you get the data structures right, the effort will make development of the rest of the program much easier [40]." The most simplistic 'model' is to only consider that which is currently being viewed and involves constantly determining if what is visible consists of traversable terrain or hazards/obstacles. This situation is comparable to a human with absolutely no short term memory. At the other end of the spectrum is building up a near perfect model, taking advantage of a robot's precision and recall. Though simplistic, the first model is highly adaptive and can easily handle random errors in the input data as well as issues such as slipping or sudden dis-/reorientation. A detailed robotic model may be able to successfully distinguish errors in input but this will require a significant amount of processing and comparison with input points around the error. The issues of orientation and position are extremely difficult for a precision model to overcome, requiring immense time to process or leading to substantial errors in judgement.

These different methods require different forms of "memory" or data storage. A current view model requires virtually no memory whereas a precision model may require large amounts of memory in the form of lists or multidimensional arrays of data.

A simple model is more robust against errors and sudden dis/reorientation and might overall use less resources (processing & memory), however a precision

model is more likely to give optimal results. The level of detail and complexity one chooses is important as it impacts on the design of the path planning algorithm in how much data it will have to base decisions upon, how adaptive it needs to be, and how robust against error it must be made.

In computer games, terrain models tend to be meshes of non-overlapping polygons as this method cuts down on the amount of memory needed to store similar amounts of detail and also lowers processing and search times. As any polygon with three or more sides can be represented as the summation of multiple triangles[41], meshes often consist solely of triangles, as the uniformity of shape aids simplicity to the design of functions or algorithms, especially in computational geometry. Using simple compact structures is also beneficial for hardware rendering of graphics.

Work with search algorithms has more commonly involved *a priori* data and often glossed over the practicalities of collating and analysing data, such that environments are frequently modelled as perfect grids with details abstracted to being traversable or not and having uniform distances between them irrespective of the affect of heights [33]. This type of modelling is beneficial as no separate network of paths needs to be generated as the model in of itself provides that functionality and in a very simple manner as only a fixed number of neighbours can exist and in preknown locations, the only variance being in whether they are considered traversable or not. However, the practicalities of attaining such a uniform/regular grid model are not realistic for a dynamic environment due to a number of factors. One such factor is the non-linear spread of angles through a lens meaning a single lens can only ensure the x or y component of data taken directly in front of it, with all other data points occurring radially from the point of view and their distance being dependent on when they encounter a rise/object in the line of sight. Regular and precise movement would also be required in order to build up a uniform grid of data points. As such, a regular grid model is best obtained in advance and from an aerial unit or extensive coverage by ground units, which removes the dynamic nature of the situation.

In addition to being incompatible with dynamic generation from a single ground

robot, there are two key drawbacks to the grid network. Firstly, the resource requirements associated with this method of modelling increase rapidly with limited/little gain in information in response. Secondly, the potential paths drawn from a grid model are also limited, with only eight directions of movement and are not optimised or specific to the presence of obstacles in different terrains.

It should be noted that the terrain model which a robot requires, representing its understanding of the world, is distinctly separate from the terrain model produced by the simulator. For the purpose of computer simulation, the simulator's terrain model replaces the real physical world.

3.4 Tessellation

Delaunay triangulation is often used as the means of turning a set of points into a mesh of triangles. Unfortunately the complexity of the mathematic processes involved, when applied to more than two dimensions, meant that an implementation could not be programmed from scratch. Suitable open source libraries could not initially be found within a reasonable time frame which meant that a less elegant method had to be coded.

The initial concept was to use points which were nearest neighbours to form triangles, however it was quickly realised that this was not a suitable approach. Creating an optimal tessellation would require discerning, for any given set of points, how many neighbouring points should be linked to a specific point and how their angular spacing should affect their choice. Thus this method would likely be almost as mathematically complex as the Delaunay approach but without the vigorous mathematical analysis and testing Delaunay's technique has received due to prominence and usage.

3.4.1 Iterative case-based tessellation

The initial attempts at the nearest neighbour based tessellation involved considering the possible combinations of triangles which could form around a single point

and how triangles could potentially overlap. This led to looking at the finite number of ways in which two triangles may interact in three dimensions, which subsequently developed into a case-based approach of iteratively going through the potential combinations such that a triangulation was formed devoid of overlap. The idea was to consider every combination of points and discard ones which encompassed another point, formed a line or slivery triangle, or overlapped the triangles bordering the cavity being tessellated, as outlined by the pseudo-code of TessellatePoints (Algorithm 3.1). Empty triangles were then compared to each other in the ProcessEmptyTriangles() function, given as pseudo-code in Algorithm 3.2, which subsequently calls CompareTriangles() (Algorithm 3.3), to choose triangles which were of a better shape and thus formed a better mesh, as well as determining whether an overlap occurred between triangles. During these comparisons triangles may have a status flag set to identifiy them as being blacklisted due to a better triangle overlapping them. TessellatePoints() calls ProcessEmptyTriangles() twice, so it can reconsider triangles which were blacklisted the first time through, to ensure no holes exist in the mesh as triangles which (through comparison) may have lead to another being blacklisted, may have later been blacklisted themselves. The quality of shape was determined based on the internal area of the triangles compared to the distance of the vertices from the centroid (equilateral triangles being the best and the worst being long thin ones). Testing for overlap between triangles was split based on the number of vertices two triangles had in common. For two points in common, the primary test was seeing if the additional point of each triangle was on the same side of the shared line, thus indicating overlap. When there were no common points between two triangles, the edges of each triangle were tested for intersections with the other. In the case of a single vertex being shared, edge intersections were also tested, comparing the edges off the vertex to the edge of the other triangle adjacent to the vertex. Each of the different overlap testing functions, would then decide which triangle to keep based on internal results or if the way the triangles intersected did not indicate a superior triangle, then size and shape were used.

The mathematics used in the different functions employed a top down, XY-

Algorithm 3.1 TessellatePoints()

```
given the new Points and the Triangles around cavity
for all coords do
  check for double ups
end for
for all (order unimportant, i.e. abc = cab) combinations of three points do
  calculate area inside three points
  if area indicates they do not form a line or triangle which is very small or
  slivery then
    for all other points do
      if another point lies inside, on the edge of or above the new triangle
      formed by three points then
        break
      end if
      for all triangles bordering cavity do
        call CompareTriangles()
        if new triangle overlaps or intersects triangle with vertex or edge
        along cavity then
          break
        end if
      end for
      if triangle is empty and doesn't overlap or intersect then
        save combination as a new triangle
      end if
    end for
  end if
end for
call ProcessEmptyTriangles() twice
call SortTriangles()
append cavity triangles to array of new triangles
call LinkTriangles() on the array of new triangles
```

Algorithm 3.2 ProcessEmptyTriangles()

```
Given the empty triangles
for triA is each emptyTri do
  if (firstRun and not blacklisted) or ( secondRun and blacklisted) then
    for first run triB is every untested tri, if second run is each whitelisted tri
    do
      call CompareTriangles()
      if aBetterThanB then
        add b to list of overlappedTri
      else if bBetterThanA then
        blacklist A
        break
      else if parallelTrans then
        set A and B as parallelTrans
      else if bothBad then
        reset list of overlappedTri to just b
        blacklist A
        break
      end if
    end for
    if firstRun then
      if aBetterThanB then
        blacklist all overlapped tri
      else if no overlap and A is not parallelTrans then
        whitelist A
      end if
    else if no overlap and not paralleTrans then
      whitelist triA
    end if
  end if
end for
if not firstRun then
  for all emptyTri do
    if whitelisted or parallelTrans then
      create Triangle object and add to newTris array
    end if
  end for
end if
```

axis approach, to identify overlaps and to judge validity. Triangles purely in the vertical plane caused numerous issues and required many small modifications to be made as new scenarios in which they altered results were detected during testing. Due to the more complex situations which could be encountered with triangles which were perfectly vertical, when encountered during comparison they were skipped unless they were being compared to another vertical triangle, in which case they were temporarily transformed to the XY-axis so tests for overlapping could occur. In all other instances vertical triangles were omitted from comparisons until the linking occurred. Vertical triangles could be erroneous, generated off the sides of obstacles and hence expanding their impact on paths, when a point at the base of the obstacle and another point higher up on the obstacle happened to form a line in the XY-plane with a point on the ground. As shown in Figure 3.3, these triangles can be identified through having no neighbour along the top (skywards facing) edge but having two neighbours on each of the other edges, with labels A and B showing the two neighbours of the pink triangle along its bottom edge. When the triangles were all being linked, vertical ones were not considered until the very end (having been placed at the end of the array by SortTriangles()) which allowed erroneous ones to be detected easily due to their existence bordering a ground triangle which already had a neighbour along the same edge. To aid recognising valid triangles the PARALLEL_VERT_TRANS status was used, which indicated that a matching triangle exists in parallel, an occurence common with cubic or prismic obstacles and uncommonly occurring with erroneous triangles.

The inclusion of software to display the triangle meshes rather than reading the lists of coordinates forming triangles and having to picture them mentally or draw them by hand, sped testing up considerably and also allowed white box testing as opposed to just the black box testing of comparing input and final output. Three dimensional displays with a rotateable view would be the best option, however for ease of implementation the Simple DirectMedia Layer (SDL) library was used to display the triangles in the view of the XY, XZ and YZ planes.

The tessellation proved a large stumbling block and mental drain, and as time

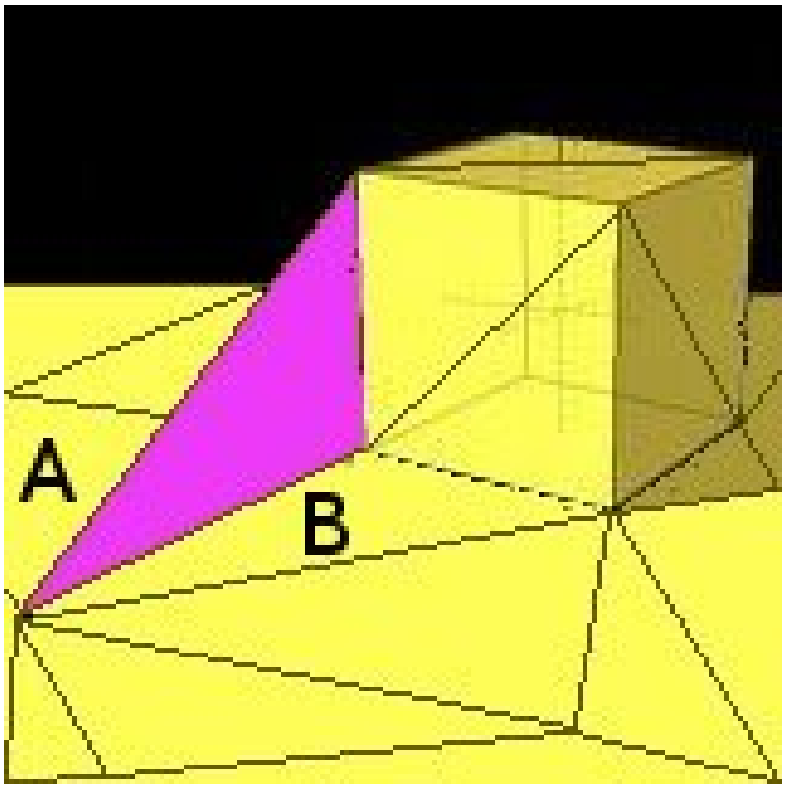

Figure 3.3: An example tessellation of a cube placed a flat plane, with good triangles shown in yellow while an erroneous triangle is coloured pink.

Algorithm 3.3 CompareTriangles()

```
Given two sets (A and B) of triangle vertices,
Calculate which points, if any, are common between the two triangles as viewed
in the XY-plane
if three shared points i.e. identical match then
  choose A over B
else
  Calculate cross produce to determine if A and B are in same (or parallel)
  plane
  if either triangle is vertical i.e. orthogonal to XY-plane then
    skip triangle
  end if
  if same or parallel plane then
    if triangles are vertical translations of each others then
      set as PARALLEL_VERT_TRANS
    end if
  else if numShared == 0 then
    check edge intersections for four of the six edge combination
    if any intersections occur then
      choose triangle which is lower, i.e. is the one being overlapped
    else
      both ok, no overlap
    end if
  else if numShared == 1 then
    check edge intersections to see if edges coming off the shared point, inter-
    sect the edge of other tri adjacent shared vertex
    if both edges of one tri cross the adjacent of other tri then
      keep the crossed tri
    else if each is crossed once then
      call CalcBetterTriangle()
    else
      both ok, no overlap
    end if
  else
    if non-shared point of each triangle is on same side of shared edge then
      if samePlane then
        pick first tri over second
      else
        call CalcBetterTriangle()
      end if
    else
      both ok, no overlap
    end if
  end if
end if
```

progressed alternative approaches were explored, such as the 'robot' being given pre-tessellated triangle meshes from the graphics engine's model of the terrain. This would remove the effects on the path planning of variance in tessellation as the simulation had very uniform meshes which perfectly modelled the terrain, thus removing some realism but allowing for completion of the project. A compromise of using the pre-tessellated triangles as the input points for the tessellation process proved fruitful as the better dispersal of points meant triangles were generally of a better shape and size, so that errors or invalid choices by the program were more obvious. Being better able to identify issues with the tessellation, the algorithm was eventually completed and was able to handle the original terrain scanning input which was more realistic.

As the tessellation had to be incremental, to minimise processing each time a new set of scanned data was received, for new points lying within known areas it was necessary to identify triangles which would be affected by the new point's addition to the terrain. A cavity within the tessellation was formed by removing these triangles which were affected. This area to be retessellated could be troublesome if it was concave, as new triangles formed through looking at all combinations of points in the cavity, may produce a triangle which partially overlaps triangles outside of the cavity. Keeping track of the triangles which bordered the cavity, in order to test newly formed triangles against them, ensured that there was no overlapping and helped to handle concave edges. Infrequent issues were still observed but with time and testing these were eventually dealt with. Though designed to approach the problem in a simple and quick manner, the iterative case-based tessellation and quest for optimal triangle meshes became a bottleneck in progress.

3.4.2 Recursive tessellation

Due to the lengthy processing time required by the Iterative Tesselation Method, ways to increase the speed were considered. At the beginning of this project, tessellation had been viewed as going directly from a set of vertices to a set of complete triangles. After having used 'Edge' objects in an attempt to deal with

the issue of convex cavities, the value of the intermediate step of creating edges from vertices, from which triangles could be composed, became clear. With this in mind the nearest neighbour approach was revisited, with speed of processing as the primary goal rather than maximising the quality of shape and size.

Starting with an arbitrarily chosen edge rather than a single point, cut down the decisions and processing as there can only be one triangle on either side of an edge as opposed to a point which could have any number of triangles formed off of it. It was felt that recursion was the best method of implementing this approach since the initial edge was branched out from - much like forming a binary search tree [42]. This proved to be much faster to code as there were few possible scenarios that could be encountered during the tessellation, with overlapping triangles being the main concern but being simple and quick to test. With edges keeping track of which sides already had triangles formed on them it was easy to check for overlaps. Also, as the tessellation expands outwards in an approximately geographical sense, it was less likely to encounter points already used, as opposed to going through points based on the input order as was the case with the previous iterative approach. This approach was much faster but often produced a number of slivery triangles in the meshes, which were hard to deal with during later stages of the path planning, such as hazard identification.

One reason why slivery triangles are difficult to work with lies in their poor shape making it hard to determine their angle with respect to the XY-plane. They are also problematic when forming compound hazards from multiple triangles. When expanding the search by moving between neighbours, reaching nearby triangles of a good size and shape may require crossing numerous slivery triangles which could be coming off a single point. This can mean a lot of processing to cover a small region, which over the entire area can added up to a very significant increase in processing requirements. Judging when to halt during this expansion across neighbours can also be problematic as the centroids of the slivery triangles may be outside the region of interest but may be bordering othr triangles of a better shape.

Adding a new point to the terrain tessellation can also take longer. When

finding the encompassing triangle of a new point, the closest existing point is found and the search begins expanding from there. Though the new and existing points might be within a short distance of each other, there could be numerous slivery triangles between them.

3.4.3 CGAL based tessellation

Upon completion of the iterative method and with the recursive tessellation going through final testing, a software library called the Computational Geometry Algorithms Library(CGAL) was found via a paper on mapping terrains, which mentioned using Delaunay triangulation [43].

This library meet all of the criteria which had initially been searched for; being open source for research/non-commercial purposes, written in C++, and doing Delaunay Triangulations in three dimensions.

Utilising this package proved quite difficult due to the vast number of functions provided by the library. Identifying which objects and functions suited the project's application was difficult. The library lacks comments and does not follow good naming conventions, which, given that the code largely consists of templates, meant it was hard to determine the purpose and usage of many objects, functions, and variables. Comprehending a single item often required searching for information scattered throughout the reference manual, which consists of over 3500 pages.

The different Triangulation objects of the CGAL library provide both two and three dimensional triangulations and come in different forms, such as regular, Delaunay or constrained. The regular triangulation simply splits the encompassing triangle when a new point is added whereas the Delaunay tries to satisfy the condition of no triangle's circumcircle encompassing another point, which may require more than just the encompassing triangle to be deconstructed. The constrained triangulations are ones which are required to retain certain edges/vectors and could be of use if future filtering was conducted on the input data to detect edges. The triangulations can be used as a mesh or placed inside a Triangulation_hierarchy object which have an internal structuring designed to allow for faster and more

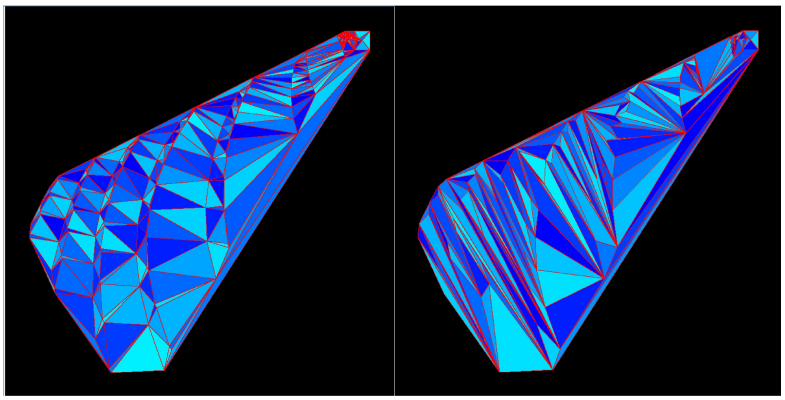

Figure 3.4: Triangle meshes generated by the CGAL (left) and Iterative (right) based tessellations, given the same series of inputs.

efficient point location.

When the resultant tessellations using CGAL were compared to the iterative and recursive approaches, they were found to be of a better overall quality and were produced significantly faster than the Iterative case-based approach. Example triangle meshes generated by the Iterative and CGAL approaches are shown in Figure 3.4, with some similarity being visible between the two but with the Iterative mesh containing numerous undesirable long thin triangles and the CGAL triangles being far more uniform in shape and arrangement.

3.5 Point Culling

The volume and quality of input data to a program is a major factor in determining the quality of the resultant output, as even the best processing can do little when only given a small number of good input values or many unreliable values. Though the precision of measurements taken by an autonomous robot whilst gathering data about its environment can affect the quality of input, this is of a fixed nature and can only be improved by upgrading to better sensors. Increasing the accuracy of sensors and attaining more data does improve the quality of output possible, however this can be costly in terms of both the hardware and the

processing time. To achieve the same high level of quality in a less expensive manner, input data needs to be dynamically condensed down to a smaller set of more valuable data through the appropriate culling of data.

With the terrain data, the preliminary processing consists of checking whether or not a new datum point lies within the known area. Points inside the known region are evaluated with regards to the triangle which encompasses them, looking first at whether the point is coplanar to the triangle or if its height is within an acceptable level of deviation from the interpolated value of the triangle. If it is outside the accuracy tolerance a point is added, and if not then the proximity of the point to other known points is assessed. The red lines in Figure 3.5 show the distances which would be calculated to determine whether point A is too close to the existing points. For a coplanar point, if any known point is within a minimum distance threshold then the point is disregarded. This means that even if the encompassing triangle is very large, the new point is only added if it is sufficiently far away from all the vertices, for example point B may not be added to the model shown in Figure 3.5. The same action occurs for non-coplanar points which are within the tolerance of accuracy however the minimum distance threshold is half as much. The reasoning for the different distance thresholds is that non-coplanar points immediately affect the terrain, possibly providing new information or possibly being inaccuracies. A coplanar point is added as a precaution, to avoid losing information and becoming more inaccurate in the case where the triangle does not accurately match the terrain. For example, a sudden rise or drop occurring within the triangle's area may appear as a gentle slope if nearer points are not included. This process takes place within the World object, specifically in the AddPoint function, which is presented as pseudo-code in Algorithm 3.4.

As a robot progresses, the value of keeping old data declines. The precise heights of a section of terrain far away from the robot has little to no impact unless back-tracking occurs. Even in cases of reapproaching previously encountered terrain, events may have occurred that have caused the data to be less accurate or perhaps even erroneous. These occurrences could include positional slips or disorientation which may have befallen the robot, or alternately the terrain having

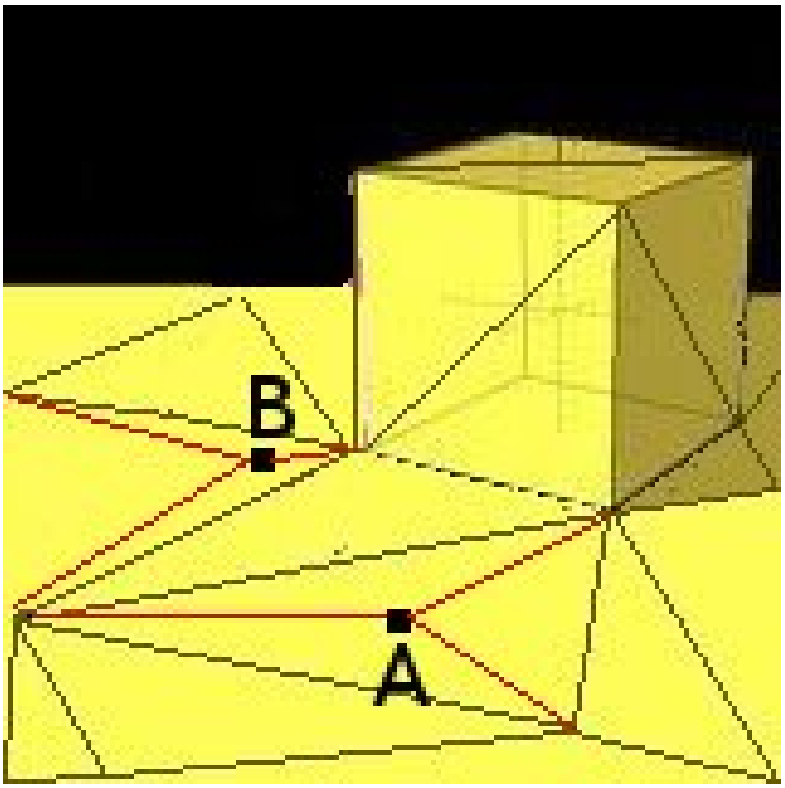

Figure 3.5: An example of comparing two new points, A and B, to the existing triangle mesh in order to determine whether to insert the new points or cull them.

Algorithm 3.4 World::AddPoint()

```
Given a new point
Check the model to see if a Face encompasses the XY component of given point
if encompassed then
  boolean addPoint is true
  interpolate Z-value of the model's surface at XY
  if interpolated value minus given value is within coplanar precision then
    check distances from point to vertices of encompassing face
    if any distance greater than threshold for coplanar points then
      addPoint is false
    end if
  else if interpolated value minus given value is within general precision then
    check distances from point to vertices of encompassing face
    if any distance greater than threshold for general points then
      addPoint is false
    end if
  end if
  if addPoint then
    model.AddPoint( Coord )
  end if
else
  model.AddPoint( Coord )
end if
```

altered due to the movement of unstable rubble or perhaps even the robot's own traversal over it. A means of limiting the resources required by old data such that it is more inline with the potential value it may serve, is to abstract the terrain to a simple dichotomy of being hazardous or traversable. Once specific height data is no longer being looked at, one can view all information about traversable regions as being superfluous as they can be defined as the regions which are not hazardous. Creating simple objects to be retained, which outline the locations of hazards and their general size and shape, while discarding the vast quantity of height data greatly reduces the data storage and processing resources used.

An alternative method of dealing with information loss can be the use of confidence levels, which alter a triangle's associated confidence value when points which lie within it are added or culled. A confidence level indicates the likelihood that the triangle is an accurate representation of the terrain covered, rather than needing to keep a dense coverage of points to feel assured. On top of this, a decay factor can be added which over time will lower the confidence level unless points reinforcing its accuracy are received. This approach is useful in dealing with errors in location or orientation as the terrain dynamically changes over time, meaning the terrain model is more robust as any mismatch between the new data and old data is slowly filtered out. This has not yet been implemented and is recommended for future work, with data points currently being removed based on distance from the robot instead, which is not as beneficial if backtracking needs to occur. For points which lie outside the known area, their proximity to known points it used as a guide to whether they should be added.

3.6 Hazard Identification

The hazards a robot faces while navigating a disaster site can be divided into the two categories of static and dynamic hazards, with dynamic ones (such as falling rubble) being harder to detect and avoid. The static hazards such as holes, cliffs, crevasses, and insurmountable inclines which a robot comes across are largely related, with the common factor being the overall slopes and relative angles, which

greatly reduces the complexity of identifying hazards. Clearances of traversable spaces between hazardous regions can also be a problem causing a robot to get jammed or be blocked, however they are not considered terrain hazards but rather navigational problems.

Dividing static hazards into, slopes which can not be ascended but could be travelled down, and walls which are completely insurmountable, allows some freedom and risk taking in the path planning, as a robot may decide to risk going down a slope it could not climb back up, in order to reach its goal. What is considered a slope or wall is determined by input parameters for each robot, consisting of what angles it is powerful enough to climb, the size of its wheels, and at which angle it would topple or roll. As long as a slope is no longer than the wheel base of the robot or if the height of a wall is less than the the wheel radius, then they can be considered as safe.

One difficult instance to classify is when a section of terrain is at an insurmountable incline, but is not big enough to be considered a wall by and in of itself, and is surrounded by what would be labelled a slope. The overall affect of the section on its surrounds has to be considered to determine whether, in combination with it, the surrounding region should be classified as a wall. There are also problems associated with triangles which are quite large, as an area much greater than a present hazard may appear to be a non-traversable incline due to the model being a poor fit. Additional data which may clarify the status of such areas is unlikely to be received as the robot will likely avoid the region, which in the event of the region being traversable is unsatisfactory as it discards potential paths. Through setting an upper size threshold (hazard_tri_threshold) above which triangles' hazard status are ignored, resolution errors are overcome. If better paths exist they are followed and it is irrelevant whether the sparse region is traversable or not. Otherwise, the region may be orientated or moved towards, and gathering further data will either repulse the robot in search of other paths or show the region to be safe to traverse.

The generalised processes involved in the hazard identification section of the World object, as split across IdentifyHazards and CheckWall, are given as pseudo-

code in Algorithms 3.5 and 3.6.

Crevasses are an extremely problematic hazard as their danger is based on relative orientation. Depending on the size of the robot's tyres, crevasses can be traversable in a direction orthogonal to that in which they run while they are likely to catch and trap the wheel if they are encountered from an incident angle closer to that direction is which they run. The difficult of dealing with this direction based danger is exacerbated by the fact that identifying crevasses from directions at which they are traversable is practically impossible at a distance, only being identifiable once very close and with sensors aimed sharply downwards such that their line of sight can judge the depth. Approaching roughly parallel to a crevasse, a sensor may be able to ascertain a significant portion of the crevasses depth but is likely to be dangerously close to having a wheel slip into the crevasse before it manages to accumulate enough data points to identify it as a hazard. Since it is not until a robot is much closer that it will be able to identify how deep a slope goes, whether it is a small dip or in fact a crevasse, this affects the quality of path planning and the safety of the robot.

3.7 Creation and linking of Node Points/Networks

The decision of which points in a terrain model are the most valuable to the path planning process and how many points to have is highly important. The number of nodes that will be used to generate a possible path network affects computation time at two points. Firstly, the point when the nodes must be tested to decide if they should be linked, and secondly, the point when the navigation algorithm analyses the possible path network to find its estimate of the best potential path. The basis on which the nodes are created is also important in terms of what the risk of encountering hazards will be, and how optimal the paths will be.

Upon developing the simulator towards implementing a greater range of algorithms, it was found that most examples given for more advanced algorithms were based on a simplified grid-based fixed-distance/resolution network of nodes as opposed to flexible unrestricted networks [33]. To allow the algorithms to run

Algorithm 3.5 World::IdentifyHazards()

```
Given the surface/region to be analysed
for each Face in Surface do
  if face already checked then
    continue For loop as already dealt with
  end if
  if internal distances (i.e. distance from vertices to centre) greater than hazard_tri_threshold then
    continue For loop
  end if
  if angle of face is steeper than SLOPE then
    if angle less than WALL steepness then
      if max dist across face divided by cosine of angle is greater than or equal to the AXLESEP then
        face is a SLOPE
        continue For loop
      end if
    else if max dist divided by sine of angle is greater than or equal to WHEELRADIUS then
      face is a WALL
      continue For loop
    end if
    for each neighbour of face do do
      if neighbour is a hazard then
        if in conjunction with neighbour, face forms part of a hazard then
          face is part_hazard
        end if
      else
        if neighbour is steeper than SLOPE but too small on its own then
          call CheckWall to see if when expanded to more neighbours they form a hazard
          if form hazard then
            face is hazardType returned by CheckWall
          end if
        end if
      end if
    end for
  end if
end for
```

Algorithm 3.6 World::CheckWall()

```
Given the initial Face, a Test_Face and HazardType status
Calculate distances of vertices to centre, for Test_Face
if (max dist is greater than HAZARD_TRI_DIST_THRESHOLD) then
  return false
end if
Calculate steepest vector formed by any two points of the two faces
Calculate dist of steepest vector and angle relative to XY-plane
if angle and dist indicate a WALL or SLOPE is formed then
  set Face and HazardType according to which is formed
  if Test_Face was not hazard or was a partial hazard then
    Set Test_Face also
  end if
else if steep enough but not long enough then
  for each neighbour of Test_Face which is unchecked do
    boolean hazards = false
    if neighbor is a WALL or SLOPE then
      Set HazardType according to angle of steepest vector between Face and
      Test_Face
    else if angle of neighbour relative to XY-plane is steep enough to be a
    hazard then
      mark neighbour as checked
      call CheckWall() with the neighbour as the Test_Face
    end if
    if hazardType has been set as a hazard then
      Test_Face is set as that hazard
      break from For-loop
    end if
  end for
end if
return true if any faces were set as hazards
```

on their originally presented grid-networks, the project was branched to allow an alternate method of node creation and linking, such that the performance of both types of path networks could be compared to identify if there were any short comings when using the algorithms on less uniform or restricted networks of potential paths. Functions and objects were created and included, but full functionality could not be implemented due to time constraints and more pressing issues/features.

3.7.1 Centroid based networks

Using the centroids of traversable triangles and linking them only to their traversable neighbours is one approach which can produce a good network of potential paths and removes the cost of testing all combinations of points for linking. The large number of nodes in the network does mean a path planning algorithm may run slowly depending on its design and also it is hard to determine if a node is actually safe to move to once one takes into account the physical dimensions of the robot. Another issue likely to arise from a network based on the centroid's of triangles is that paths will zig-zag since a straight line of centroids through multiple adjacent triangles is unlikely. If the mesh was created from other polygons such as squares or rectangular, the path between two unobstructed points would be less likely to oscillate. However, as outlined already, triangles provide the greatest overall benefit for modelling and as such alternate methods of network creation which are compatible with triangle meshes are advisable.

3.7.2 Border based network

Alternative methods were considered which focussed on the border between hazardous and traversable triangles. Linking the midpoints of the edges which create this boundary explicitly allows the robot to know where a hazard is and thus to avoid such nodes by a buffer distance. However, calculations of distances between nodes which some algorithms may do as part of their analysis will have varying levels of inaccuracy when compared to actual path distances the robot may travel.

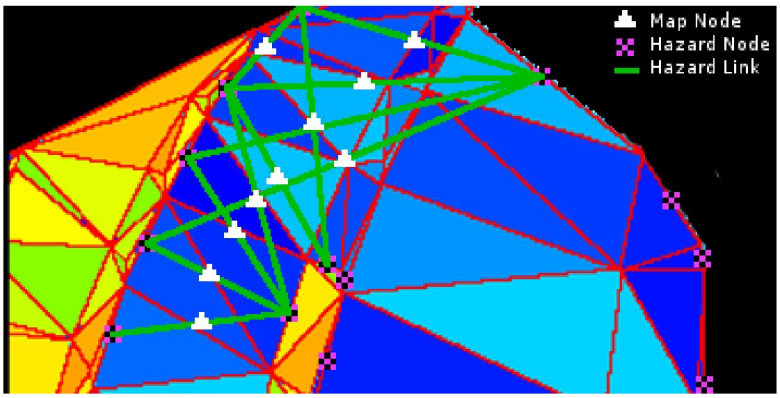

Figure 3.6: Example of Hazard and Map Node creation, showing only a portion of the potential links. The blue triangles being traversable and all others colours being hazards.

Creating nodes a set buffer distance from the border midpoints overcomes this inaccuracy but decreases certainty that a hazard will not be encountered as the node offset from one edge may cross the path/offset of another node and become close to a nearby hazard edge. The method which was chosen involves trying to place the map nodes in the middle of clear regions of terrain, and thus also looks at edges bordering regions which are as yet unknown. Clear central points are found through taking all the edge midpoints, trying to link them and looking at the distance the link covers. If the distance is over a certain threshold then the mid point of the link is used to create a map node. The mid points of these hazard/unknown links can be quite close to each other when links cross, so the set of map nodes needs to be checked for their relative proximity in order to discard superfluous nodes. Figure 3.6 helps to illustrate this point, showing a terrain which has hazards along the left and in the bottom centre while the right side is the limit of the known terrain. Even with only half the links from which map nodes can be created being displayed, overlapping occurs frequently, often placing midpoints beside each other. Once filtered down, the map nodes are then all linked together, which means the process of linking nodes is performed twice. However, this is on a relatively low number of points and provides a more robust network of potential paths.

3.7.3 Fixed resolution grids

While fixed grid node networks are often simply derived from having a grid based terrain model double as the node network, they can also be easily generated for other types of models. To begin, with the initial position and an orientation vector are chosen, often the robot's location and either north, the direction the robot is facing or the direction of the destination is specified. A grid network is formed by sampling the heights for set XY-coordinates, which are separated by fixed increments in four directions (up, right, down and left) (0, 90, 180 and 270 degrees relative to the direction vector). With nodes being uniformly distributed, they can be linked to either 4 or 8 neighbouring nodes, dependant on whether diagonal moves/links are to be included/allowed.

Much like using fixed resolution grids for terrain modelling, this method of network creation is fairly simple but does not produce results specific to the position of obstacles. Consequently, fixed grids may miss potential paths between obstacles due to alignment of the grid with the obstacles and the clear section, with how this may occur being demonstrated in Figure 3.7, as a slight shift of the obstacles with respect to the grid points, resulting in more points being classified as traversable (green) as opposed to hazardous (red).

3.7.4 Occupancy grids

An enhancement on using fixed resolution grids is to move from solely using the vertices of where the grid lines intersect to considering the area within square sections of the grid and then using the centre of these squares as the nodes. The terrain bounded by these square sections can be analysed to produce a probability of being hazardous/not traversable based on the proportion of area which is classified as such. This probability can be considered a risk-value, which can be used by algorithms to shape their behaviour in terms of weighting different links/paths based on perceived risk. A threshold below which they may consider a section traversable could also be used and as such negate issues with the alignment of the grid with respect to the position of paths through obstacles. This basis of using

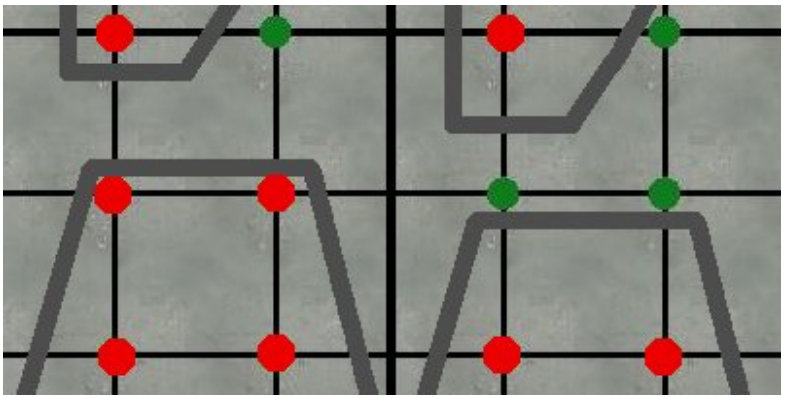

Figure 3.7: An example terrain model of a fixed resolution grid, where green nodes are traversable and red ones are not. Given the boundaries of hazard obstacles, shown as grey lines, the arbitrary alignment of the grid with respect to the terrain may produce different models, with the left one having fewer traversable points for the same area of terrain.

probabilities and thresholds is also beneficial in dealing with noise introduced by input sensors and errors caused through inaccurate positioning, similar to the use of confidence & decay levels already mentioned.

For display purposes probabilities can be depicted in greyscale, with black indicating the definite presence of an obstacle, white indicating no obstacle, and the lighter or darker shades of grey indicating a lower or higher likelihood of an obstacle existing at that location. Figure 3.8 presents an example of occupancy grids being used to create maps of a unit's view from specific locations.

3.8 Data Objects

3.8.1 Coords, Triangles & Edges

The initial Abstract Data Type (ADT) used to represent directions and positions was called a Coord. Coords held x, y and z coordinates and had a number of associated mathematical functions. In addition to the basic mathematical functions such as addition and subtraction, Coords also had Cross Product, Dot Product

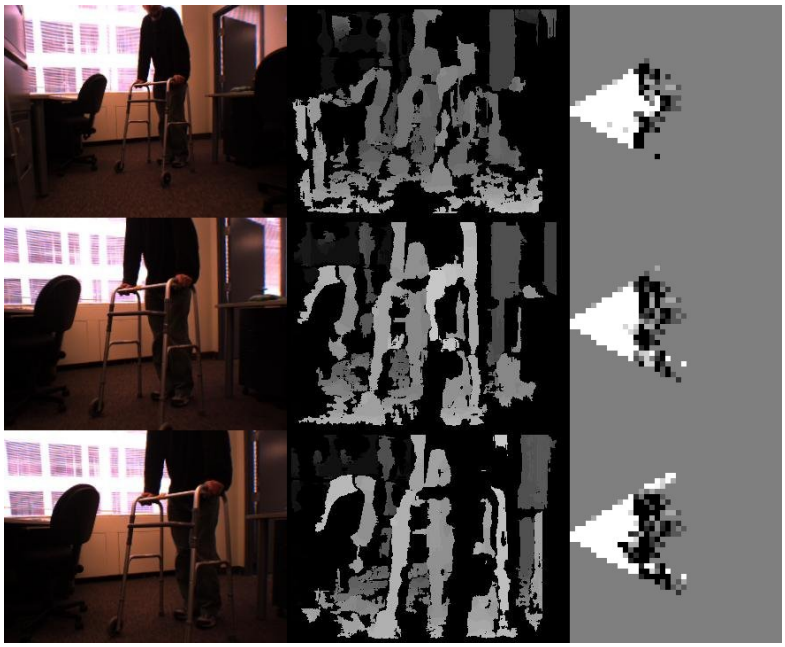

Figure 3.8: Example of Occupancy Grids in use by a wheelchair collision avoidance system approaching an elderly man with a walking frame [44].

and a number of comparative functions. The next layer of objects were Triangles which formed the mesh representing the terrain. Triangles contained links to three Coords which represented the Triangle's vertices as well as links to the three other Triangles which neighboured it. Most of the object functions for Triangles involved comparison to other triangles, to check for points or neighbours in common or if they were different instances of the same Triangle.

Coords and Triangles acted as one of the base underlying objects throughout the design stages of both the iterative and recursive tessellation, with Edges only being utilised in the recursive approach. As one would expect, Edges had two Coords however they only had one neighbouring Triangle as only the first one was of import during the tessellation process. To identify whether triangles had been created on either side of the edge there were two Booleans, these also acted to identify which side the neighbouring triangle lay on when only one had yet been created.

3.8.2 Points & Face_handles

Using CGAL to do the terrain modelling required that a number of other objects types were used in order to successfully interface/interact with the triangulation it created. The primary objects for CGAL triangulations are Points and Face_handles, which are analogous to the previously outlined Coords and Triangles. To accommodate these objects, conversion functions were created, as well as many other functions being templated to allow either Coords and Triangles or Points and Face_handles to be used. Modification of the Coord and Triangle objects so that variables and functions had names matching their CGAL counterparts was also necessary, in order to use either type in template functions. Face_handles were also altered to include the hazard status of the face/triangle.

3.8.3 Nodes

The potential path network on which path planning occurs, consists of numerous interlinked Node objects. The network can consist of either MapNodes or ASTAR_MapNodes, which are each associated with different types of path planning algorithms. However, before the network is created HazardNodes are used as an intermediary stage in determining the positions of MapNodes or ASTAR_MapNodes.

Though the simulator is not fully capable of supporting grid based networks at this point in time, GridNodes and ASTAR_GridNodes ADTs were created in parallel to MapNodes and ASTAR_MapNodes, looking ahead to the eventual use in grid based networks. A UML diagram depicting the hierarchy of the Nodes is shown in Figure 3.9.

HazardNode

The faces comprising the CGAL terrain mesh are processed to identify which ones are within visible distance of the robot's current position, as these are the ones which may have been affected by the latest set of terrain/scan input data. This set of faces is then further filtered to identify hazardous faces with traversable neighbours, thus establishing the boundaries of the traversable regions. HazardNodes

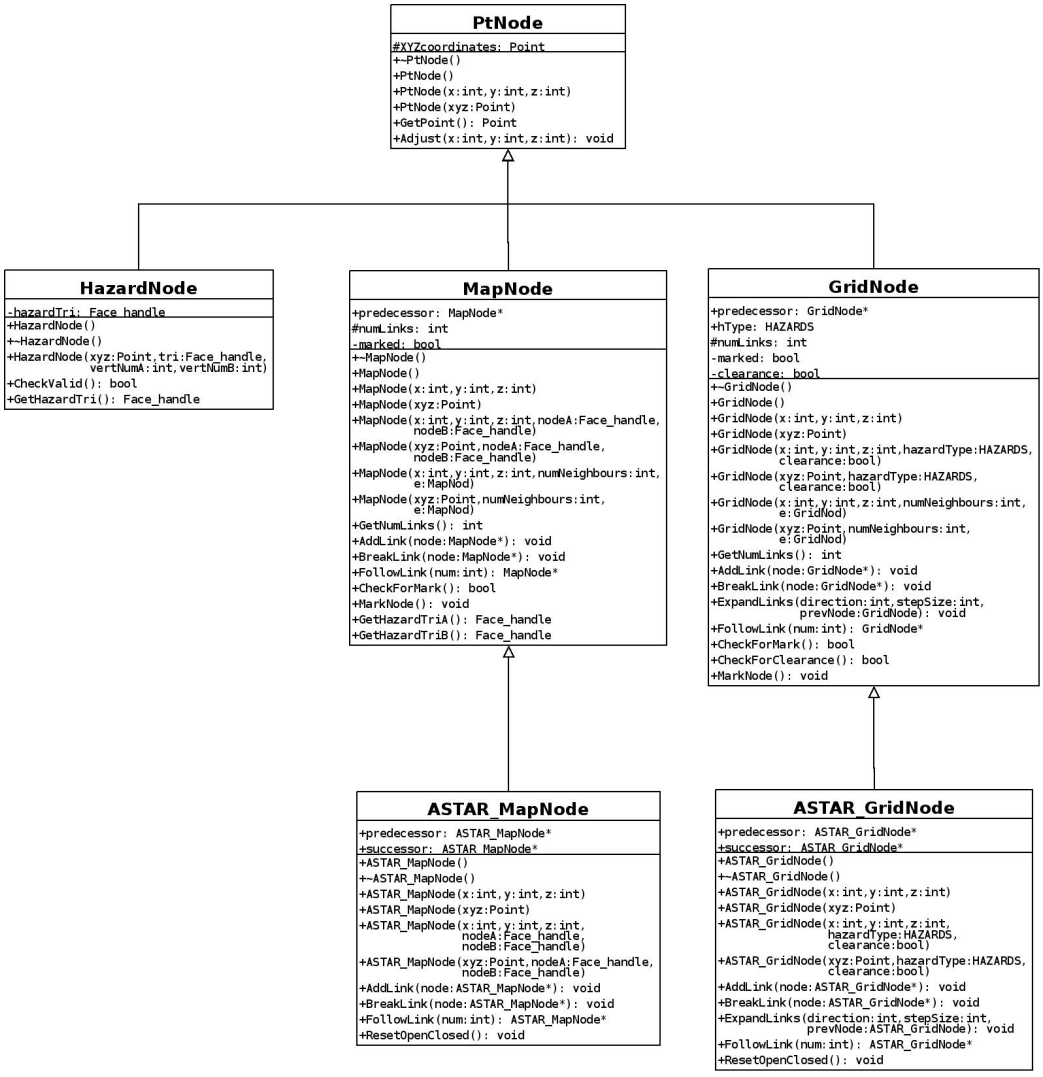

Figure 3.9: A hierarchy tree representation of the relationship between the Node objects

are created at the mid-point of the shared edges and store pointers to the points at either end of the edge/border.

MapNode

All combination of HazardNodes are checked to see if they have a clear path between them which is at least wide enough to accommodate the robot passing between them. If it is determined to met the criteria, a new MapNode is created at the mid point between the two HazardNodes.

ASTAR_MapNode

An ASTAR_MapNode is created through the same process as a MapNode and are in most ways identical to MapNodes, however they differ in what variables are included for use by path planning algorithms. The ASTAR_MapNodes are used by the A* algorithm and its derivatives, including f, g and h-value variables used by the algorithms (see 3.9.4).

3.9 Path Planning Algorithms

Building upon the tessellation and data structures, the implementation of a number of path planning algorithms was undertaken. The complexity of these algorithms varied greatly, as did the success in implementation. A great deal of difficultly was encountered with the LPA* and D* Lite algorithms since all of the information regarding the specifics of their implementations was based on using an eight direction grid-based network of nodes, an approach which had been discounted in this project due to being sub-optimal in terms of resource requirements. The core of the problems lay in the updating of the network on successive iterations. This action does not occur with the A* algorithm as it recomputes the entire network upon each iteration.

3.9.1 Edge Hugger

The first algorithm implemented was a simple edge hugger. This approach does not require the same level of processing as the rest of the algorithms as it does not require a network of potential paths to be created and instead relies solely on collision detection. Collision detection being the analysis of whether moving forward in a certain direction would cause the robot to collide with a hazard, thus allowing the robot to choose an alternate direction, parallel to the edge of the hazard to be skirted. This algorithm provided great benefit in not only testing the collision detection function but also the hazard identification process which the collision detection relied on to have analysed and identified the hazards which it would be attempting to avoid.

As the path planning system progressed and the design changed, obsolete and incompatible functions were removed or altered. The shift from points and triangles to coords and faces, along with implementation of the node network creation and linking, led to the Edge Hugger no longer being compatible with the system and as such was not used further for any experimental trials.

3.9.2 State model

A simplistic state-based path planning function was produced to provide edge-hugger/wall follower like behaviour while using the node network. The states chosen were Forward, Right, Back and Left, with each being the direction relative to facing/centring on the goal. An additional state controlled how the set of states were progressed through, either going in a clockwise (Forward, Right, Back, Left) or anti-clockwise (Forward, Left, Back, Right) manner. This ability to alternate between clockwise and anti-clockwise exploration was added to allow the robot to break free from being trapped in enclosed corners/sections/regions in which paths may be ignored due to favouring left over right or vice-versa. While not overly intelligent this algorithm intrinsically deals with dead ends and the need for back tracking/move away from the goal, in order to reach it.

3.9.3 Breadth First Search and Depth First Search

The very basic search algorithms of Breadth First (BFS) and Depth First (DFS) were considered for usage and comparison within the system. The BFS algorithm places all newly encountered nodes into an array, considering and expanding the search from the oldest entries first, which means that all neighbours of a node are considered before moving on to their neighbours. The DFS algorithm adds nodes in the same manner as the BFS, however it investigates the newest entries first, which leads to it exhausting a branch of searching before considering the neighbours. However, upon implementing and testing Depth First Search it was realised that for open unbounded scenarios the algorithm would continue on regardless of the goal location. Being a blind search algorithm there was no heuristic to draw it towards the goal and hence, without a maze-like bounding box and finite/discrete positions, there was little chance of ever reaching the goal. While a Breadth First Search may eventually find the goal, through its nature of fanning out in all directions, it would take a great deal of time and require vast amounts of resources when there was no bounding/restricting border/box.

3.9.4 A* Algorithm

Starting at the node representing the current location of the robot, the A* algorithm investigates each of the linked nodes/neighbours and calculates their f, g and h-values and sets the current node as their predecessor, before placing them into the open set and the current node into the closed set. The g-values are the known distances it is necessary to travel in order to reach any given node from the starting position while only travelling along nodal links. The direct calculated Cartesian distance between any given node and the goal is a node's h-value and is where weighting may be applied to affect the expanse/scope of investigation. Summing both the g and h-value for a node gives its f-value, which is used by the algorithm to judge the potential likelihood of being an optimal path.

The open and closed sets represent the nodes which have been already investigated, with those in the open set still having potential while the closed set contains

ones which have had all their links fully exhausted. The nodes within the open set are sorted relative to their f-values so that they are checked relative to their merit as opposed to a first-in first-out or last-in first-out strategy as applied by BFS or DFS, respectively. With the current node being fully expanded and placed in the closed set, a new node is taken from the open set and the process is repeated until the goal is reached or all potential paths have been exhausted. The current pick of the nodes may have links to nodes which are already in the open or closed sets. Nodes in the closed set can be ignored however those in the open set are re-evaluated to see if having the current choice as their predecessor would lower the nodes g-value, in which case they are relinked and have their values adjusted accordingly.

In order to deal with cases where no path is found, on each loop through the path analysis, the current pick is compared to previous ones in order to determine which node has the lowest h-value and hence is closest to the goal. A pointer to the closest node is kept such that when no path to the goal is found, it can be used as a temporary destination in the hopes further traversal will increase knowledge of the terrain and open a path to the robot.

Starting at the goal (or closest node), the predecessor is taken and placed in an array, as is the predecessor's predecessor and so on until the path between location and destination is reconstructed and held within the array. The direction of the next move is calculated and the robot passes this on to the simulator which updates the robot's position.

The assumption within A* that nodes have a clear link to the goal node unless proven otherwise was found to be problematic in this implementation, especially due to the potential for criss-crossing links. For instances where the goal node was significantly far away, the process of confirming whether hazards existed between the goal and a specific node was very intensive and also problematic when regions of unknown terrain were encountered as the discontinuity was incompatible with the confirmation technique which involved moving through neighbours. The solution used, was to only consider linking the goal node to other nodes which were within a set/visible distance of it. This approach meant the process was quicker

and it was less likely that while traversing a link it would be found that the terrain had altered significantly and required reanalysis/linking.

Due to points not automatically existing in certain positions and not being able to assume their presence outside the visible region, as occurs with the fixed grid approach, two temporary points were necessary to ensure the algorithms were presented with paths around obstacles even when very little data had been gathered for the region. These points were taken as the limits of the visible region in terms of distance and angle from the centre to both the left and right. Upon traversing either way around an obstacle the point would become unnecessary as new data would be gained through movement in that direction.

An attempt was made to implement a version of the A* algorithm which was restricted to eight directions of movement, however the grid network was not at a reliable state to support the implementation.

Algorithm 3.7 A_Star_Ctrl()

```
Given pointer to robot
if robot has not moved sufficient increments for reanalysis and has not reached
the next node in path then
    skip
else
    retrieve the node network
    call A_Star_Eval()
    go through path from goal back to current position, put coordination from
    nodes in an array
    call robot's SetPath() with an array of the path coordinates
end if
```

3.9.5 LPA* Algorithm

The previously implemented A* algorithm was taken as a base for the LPA* algorithm, being modified to retain the Open and Closed sets across multiple iterations and with functions added to identify changes between iterations and update the successors or predecessors as required. Unlike grid-based networks where nodes

Algorithm 3.8 A_Star_Eval()

```
Given the network nodes and a pointer to robot
update values of start/current position
set currentpick as start node/current position
while first loop, or open set is not empty and goal not reached yet do
  for all nodes linked to by currentPick do
    if linked node is flagged as closed then
      continue
    else if node has not been visited/analyse yet, is not in open set then
      set node as open and currentPick as predecessor
      add node to open set
      set g-value as predecessors g-value plus distance between node and pre-
      decessor
      set h-value as distance to goal
      set f-value as sum of g-value and h-value
    else
      if g-value would be lower with currentPick as predecessor then
        adjust values and set predecessor as currentPick
      end if
    end if
  end for
  sort open set in order of f-values
  if open set is not empty then
    choose node with lowest f-value as new currentPick
    move new pick from open set to closed set
    if new pick has lowest h-value yet then
      set new pick as closest
    end if
  end if
end while
if goal reached then
  temp as goal
else
  Warn is incomplete path
  set temp as closest
end if
reset/remove nodes in open set
reset/remove nodes in closed set
return temp
```

always exist and may only change in terms of their status as being traversable or not, the flexible system used as the primary form of node networks resulted in some nodes no longer existing upon a successive iteration. This facet of flexible networks caused some additional difficultly to the updating of the links as propagating a change could encounter a disconnect in a previous path due to a node's removal. Successfully determining a new predecessor or successor requires that all nodes upstream from the node being relinked have been updated and changes fully propagated, otherwise some nodes may appear locally inconsistent and be relinked when unnecessary. Incorrectly pointing to a node as the successor of another, could lead to cyclic paths when other updates were propagated due to incorrect f-values being compared.

Algorithm 3.9 LPA_Star()

```
Given pointer to robot
if robot has not moved sufficient increments for reanalysis and has not reached
the next node in path then
   skip
else
   retrieve the node network
   if Goal has no predecessor or a node in planned path is missing predecessor
   then
      call UpdateLPA() to re-calculate the planned path
   end if
end if
```

3.9.6 D* and D* Lite Algorithms

Due to the superior performance of D* Lite, as outlined by Koenig[35], in terms of having fewer tie-break conditions and being a re-use rather than re-plan algorithm, the implementation of D* was forgone in favour of D* Lite. Upon delving into how the D* Lite algorithm works, it was found to be a little too complex to implement for the flexible node network without greater familiarity with the mathematical basis and principles of the algorithm. The idea of modifying an existing implementation of the algorithm was investigated, however upon downloading a

Algorithm 3.10 UpdateLPA()

```
Given the network nodes and a pointer to robot
Retrieve open and closed sets
update values of start/current position
set currentpick as start node/current position
call UpdateSetsAndClear() to update nodes in open and closed sets then empty
them.
while first loop, or open set is not empty and goal not reached yet do
  for all nodes linked to by currentPick do
    if linked node is flagged as closed then
      continue
    else if node has not been visited/analyse yet, is not in open set then
      set node as open and currentPick as predecessor
      add node to open set
      set g-value as predecessors g-value plus distance between node and pre-
      decessor
      set h-value as distance to goal
      set f-value as sum of g-value and h-value
    else
      if g-value would be lower with currentPick as predecessor then
        adjust values and set predecessor as currentPick
        call AdjustSuccessors() to adjust f and g-values of nodes which have
        this node as predecessor
      end if
    end if
  end for
  sort open set in order of f-values
  if open set is not empty then
    choose node with lowest f-value as new currentPick
    check predecessors linked correctly (omit this as is house-keeping rather
    than algorithm?)
    move new pick from open set to closed set
    if new pick has lowest h-value yet then
      set new pick as closest
    end if
  end if
end while
if goal reached then
  temp as goal
else
  Warn is incomplete path
  set temp as closest
end if
go through path from goal back to current position, put coordination from nodes
in an array
call robot's SetPath() with an array of the path coordinates
```

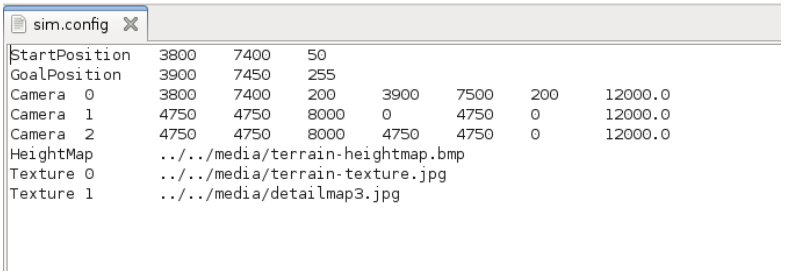

Figure 3.10: The contents of a simulation/environment configuration file. The first three columns of numbers being Cartesian coordinates for positions and the additional three columns associated with the cameras being the target positions at which to be aimed.

couple of implementations of the algorithm, it was realised this still required a better understanding of the algorithm as the underlying data structures between the current system and downloaded algorithm would need to be interfaced and altered.

3.10 Automation and Configuration

As with all scientific investigation, repetition is an important factor which adds confidence to findings. Automation of experiments is beneficial to researchers as it provides the ability to repeat trials with little effort, usually no more than entering the number of times to be repeated, and allows them to focus their attention elsewhere.

To further enhance data generating capabilities, the use of configuration files allows a program to switch between running trials that have setups which may differ substantially, with minimal effort on the operator's behalf. After the initial creation of configuration files, which are often just copies of other configuration files with some parameters altered, changing the filepath/name inputted to a simulator is all that is required to run various trials. The automated processes and configuration files allow the program to be left for extended periods, during which it will to go through various trials.

obstacles3.config

```
<Obstacle>
Position        4200    7800    4000
Rotation        0       0       0
Scale           100     100     100
Mesh            ../media/tall_box.3ds
Texture         ../media/wall.bmp
</Obstacle>

<Obstacle>
Position        3600    5900    4000
Rotation        0       0       60
Scale           50      300     100
Mesh            ../media/tall_box.3ds
Texture         ../media/wall.bmp
</Obstacle>

<Obstacle>
Position        3200    8000    4000
Rotation        0       0       0
Scale           300     50      100
Mesh            ../media/3steps.3ds
Texture         ../media/wall.bmp
</Obstacle>

<Obstacle>
Position        1600    6000    4000
Rotation        0       0       135
Scale           75      75      75
Mesh            ../media/5steps.3ds
Texture         ../media/wall.bmp
</Obstacle>
```

Figure 3.11: An obstacle configuration file for adding four obstacles to a terrain.

config_combos.txt

```
../sim.config    ../obstacles.config
../sim.config    ../obstacles1.config
../sim1.config   ../obstacles.config
../sim1.config   ../obstacles1.config
```

Figure 3.12: Four combinations of Obstacle and Simulation configurations inside a control file, allowing two different obstacles files to be each used with two simulation configurations.

The simulator created for this project uses three configuration files, with examples given in Figures 3.10, 3.11 and 3.12. The first two configuration files provide Environment/Simulator settings (Figure 3.10) and Obstacle settings (Figure 3.11) while the third file (Figure 3.12) controls which specific files are used for those configurations. The Environment settings provide information required by the graphics engine to produce the simulation, such as which heightmaps are to be used, the position and orientation of cameras, where the robot model should initially be placed and where to put the goal marker. The position, orientation and scale of obstacles to be added to the terrain are outlined in the obstacle's configuration file, along with the filepaths of which models are to be used as the obstacles.

3.11 Path Analysis

To aid both later analysis of results as well as confirming correct behaviour in initial testing it is important to have methods to display and demonstrate the results, as they can be difficult to comprehend in numerical form only. As such, a path viewing module which displays overlaid paths was created. Figure 3.13 shows the results from two sets of trials, one set from a simple terrain and one containing many more obstacles and slopes near the limit of the robots traversal capabilities. Another module has also been implemented which allows a set of movements to be replayed. This replay function runs significantly faster and hence is better suited to viewing than when the robot is actively path planning. Replays simply re-enact the movements while not having to scan the environment, analyse the terrain or calculate paths, which are the time consuming steps of the simulation.

To ensure the data to be displayed always fit in the display/viewing window, while showing as much detail as possible, the display is scaled both in the X and Y axis based on the bounding box of the points given. The relationship between the X and Y axis varies when different sets of data are displayed and as such displayed paths may appear stretched in one direction; this must be kept in mind when comparing paths to the terrain.

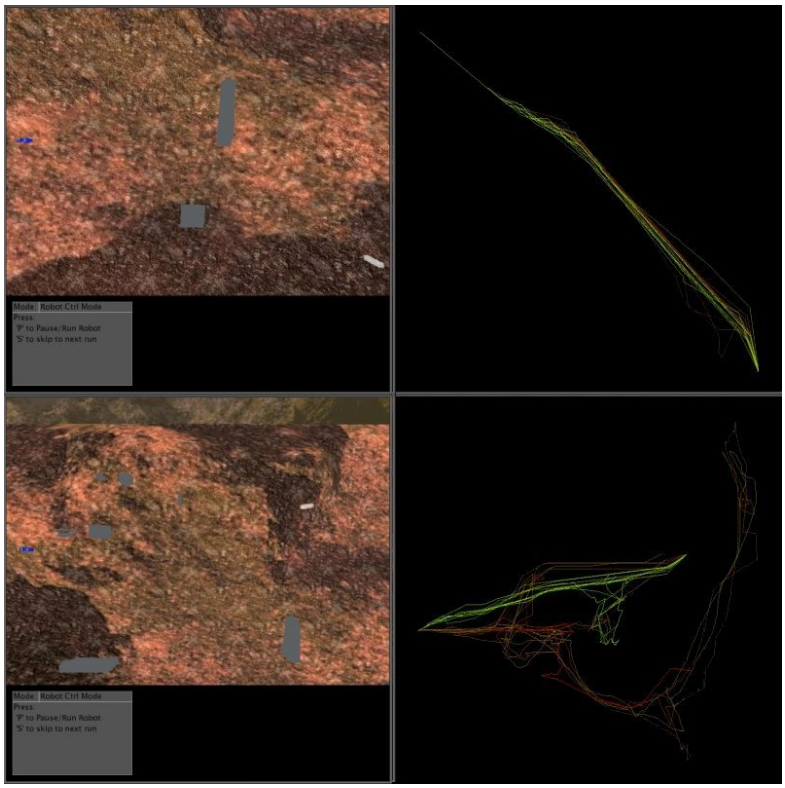

Figure 3.13: Two test scenarios, consisting of gentle slopes and two obstacles (top) and steeper slopes, numerous obstacles and an unreachable goal (bottom), displayed beside the overlaid paths generated from trials conducted on them using the A* algorithm. The point objects are the robot, the white are the goals and the grey objects are additional obstacles.

Chapter 4

Experimental Design and Pilot Tests of Path Planning

Simulations with varied robot parameters and terrain configurations were run in order to evaluate the Simulator and show that is was capable of being used for experimental investigation. In particular that the Simulator is especially useful to employ on prolonged simulations, where a variety of options are available to alter and that it produces results in a usable form and with sufficient information about the simulated trials so as to identify variance and be able to draw conclusions. Pilot testing was carried out to ascertain whether the design and setup chosen to run experimental trials was viable, to identify and correct any issues before extensive trial runs, and check that the output of the simulator was sufficient for scientific analysis.

4.1 Hardware

The computer on which pilot trials were run, had an Intel Core2 Duo 6400 processor at 2.13 GHz and 2.0 GB of RAM. The operating system used was release 5.0.2 (Lenny) of Debian, the display manager used was GNOME 2.22.3, and they were running on the Linux 2.6.26-2-686 kernel. Creating the executables the GNU C++ compiler (g++) version 4:4.3.2-2 was used, along with the C standard library

version 2.7-18, MESA 3D (an open-source computer graphics library providing the OpenGL implementation) version 7.0.3-7, the Irrlicht graphics engine version 1.5, CGAL version 3.3.1-4 and SDL gfx-package version 2.0.13-4 (equiv of SDL 1.2.13-4).

For the extended trials, eight computers in the undergraduate computer lab were initially used, but later a further four also ran simulations. These computers were chosen as they were unused over the holiday period and because the graphical requirements of the simulator excluded it from being able to run on the postgraduate computing cluster. Each computer, had an Intel Core2 Quad 6400 processor at 2.40 GHz and 3.9 GB of RAM. The operating system used was release 10 (Cambridge) of Fedora, the display manager used was GNOME 2.24.3 and they were running on the Linux 2.6.27-12 kernel. Creating the executables the GNU C++ compiler (g++) version 4:4.3.2-2 was used, along with the C standard library version 2.7-18, MESA 3D version 7.0.3-7, the Irrlicht graphics engine version 1.5 and CGAL version 3.3.1-4. SDL was not present on the computers and thus some features were omitted, however the omitted features are not needed to generate results, only being for graphically displaying information upon operator request. As the home directory for these computers is on a network drive, results were recorded to the local /tmp/ drive in order to avoid excessive overworking of the input/output of the home drive with the high load of multiple computers writing to it for an extended time period.

4.2 Variables

4.2.1 Simulator

The quantity of points within the emulation of a LIDAR scan is one of the major variables available to alter, along with the dispersal and angle of these points.

4.2.2 Algorithms

At present the only variable parameter relating to the algorithms is the float H_WEIGHTING, which affects A* based algorithms and alters the weighting of the h-values in producing f-values.

4.2.3 Robot

The variables which reflect the physical properties of the robot are the integers ROBOTLENGTH_MM, ROBOTWIDTH_MM, AXLESEP_MM (distance between axles), WHEELWIDTH_MM, WHEELRADIUS_MM and CAMERA_HEIGHT (height of the camera off the ground) plus VISIBLE_DIST_MM (maximum distance LIDAR can view) which is a float.

Characteristics of the robot's interaction with the world are defined by hazard avoidance parameters, movement parameters and algorithm choice parameters. The SLOPEANGLE and WALLANGLE are used to categorise the gradient of surfaces while CLEARANCEBUFFER_MM stipulates the clearance from hazards to be observed for safety purposes. The robot's movement is controlled by the movement step size and maximum number of movements between the algorithm reanalysing the potential path network, which are MOVE_INCREMENTS_MM and STEPS_BETWEEN_EVAL, respectively. The choice of algorithm is set within the robot object and ALG_NODE_TYPE defines which node type should be used.

4.2.4 World

Within the world object point culling is controlled by MEASUREMENT_PRECISION and DIST_THRESHOLD for general additions, while COPLANAR_PRECISION and COPLANAR_DIST_THRESHOLD cover instances where the point is coplanar with its encompassing triangle. Other variables are HAZARD_TRI_DIST_THRESHOLD (Maximum internal distance to centre for which a hazard triangle can be reliably trusted), MAPNODE_PROXIMITY_THRESHOLD, KEY_RATIO (Distance past visible region which nodes should still be held on to) and MIN_INCIDENT_ANGLE

(Minimum angle of incidence for links between hazardNodes and the border on which a hazardNode lies).

4.3 Procedure

The parameters to be altered during testing were recorded and the trials were given designations/reference codes to indicate which variable was being altered and to what value. For a set of trials, the variable of interest would be set in the source code and then an executable file compiled. From the the commandline the simulation executable would be called and passed a configuration file containing the settings of various scenarios, on which the specific set of trials should be tested. When it was desired to run multiple executables, shell scripts were written to iterate through calling the executables and to feed them the same configuration file.

As the computer on which the trials were being run was also needed for other purposes, the 'i' key was pressed to set the simulator to continue running even when it was not the active window. Upon completion of a set of trials the resultant files would be moved into a new appropriately directory, either manually or by the shell script if used, so that the results from new sets did not get appended to the past results.

4.4 Analysis

The results as listed in SimResults.txt were grouped by scenario and the termination status were tabulated to give an indication of how the different sets performed. In some cases the steps taken were compared by scanning over the outputted steps the robot had taken, but this was generally only of use when the trials ended early, otherwise graphically overlaying the paths was easier and was a greater aid in understanding the overall behaviour of a set of trials. To give a faster indication of how a set fared, for the experimental results histograms were favoured over tables, with the precise values of statistical information such as the median or mean being

generated and available if needed but otherwise left up to the charts to convey.

One factor which was considered of interest but did not end up being specifically compared was the time taken to complete the trials. The reason this characteristic was not analysed was due to an inability to regulate or monitor the varying loads on the computer from other tasks, the state of network connections and other factors not related to the simulation, which may affect the time taken.

4.5 Pilot Tests

For the pilot testing, each configuration file being considered for use in experimental trials and each parameter to be altered and its associated set of values, was trialled in order to confirm correct operation and detect any changes which may be necessary. Ten repetitions were run for each pair of terrain and obstacle settings, resulting in twenty trials for each weighting tested. The majority of trials were run using the A* algorithm due to greater confidence in the version of it ported to be used on the flexible node network, over that of the modified LPA* algorithm. It was felt the LPA* algorithm required confirmation of correct behaviour and validity, through comparison of performance on both a grid-based node network and flexible node network but the ability to create and use a grid-based network was not completed. Hence it was felt conclusions drawn from the A* algorithm would be more reliable. An overview of how the different trial sets are inter-related is shown in Figure 4.1, with the three algorithms along the top and solid arrows denoting a subset which involved a variation of parameters, while the dotted arrows indicate some similarity between sets.

Depending on which parameters were varied and to which values, on which scenario they were run, as well as the level of success and number of steps before failures, between 15 and 20 trial runs were usually completed in a 24 hour period. The theme across parameters chosen to investigate was vision and movement, as it was expected these may show some of the strongest interactions. With regards to the scenarios created for the robot to deal with, the choices of start positions, goal positions and obstacle locations were largely arbitrary and chosen to provide

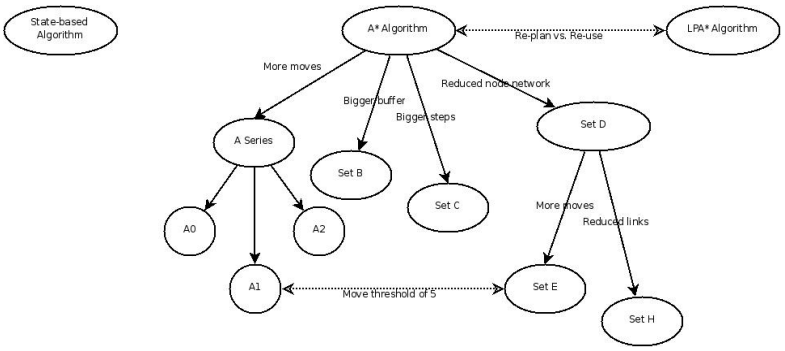

Figure 4.1: UML diagram showing the branching of different trial sets.

some variety.

4.5.1 A* algorithm

Config_combos_0

The first set of trials conducted used the A* algorithm and looked at the impact of varying the weighting of the h-value used by the algorithm. Trials were conducted on four different combinations of locations and obstacles, using the config_combos_0 configuration file, with each run occurring ten times. The terrain heightmap used by this configuration file (and config_combos_2) was one provided with the irrlicht graphics engine, being chosen due to its size and the reasonable level of variety within the terrain produced. The values of the altered h-value weighting were 1.0, 1.05, 1.1, 1.15, 1.2, 1.3, 1.4, 1.5 and 2.0, with Table 4.1 listing the completion status of each set of trials. For the first two scenarios within the configuration the results across all weightings, except for 2.0, were not significantly different/varied little. This lack of variance is not surprising considering the simple nature of these scenarios which were expected to have minimal hazard avoidance required. The other two scenarios within Config_combos_0 were highly unsuccessful in comparison, as indicated in Tables 4.2 and 4.3. It occurred frequently that the robot, for the second pair of scenarios, encountered a hazard or reached a point from which it was not able to determine a potential path to the

	Success	Failure	
H-value weighting	Goal reached	No path to goal	Hazard traversed
1.0	20	0	0
1.05	20	0	0
1.1	20	0	0
1.15	20	0	0
1.2	20	0	0
1.3	20	0	0
1.4	20	0	0
1.5	20	0	0
2.0	0	20	0

Table 4.1: Combined results of first two configurations within Config_combos_0.

	Success	Failure	
H-value weighting	Goal reached	No path to goal	Hazard traversed
1.0	8	2	0
1.05	0	1	9
1.1	0	2	8
1.15	0	3	7
1.2	0	3	7
1.3	1	1	8
1.4	0	6	4
1.5	0	10	0
2.0	0	10	0

Table 4.2: Results from third configuration within Config_combos_0.

goal. In the case of the final set some trials were not completed due to errors.

The first pair of scenarios within Config_combos_0 consisted of a flat terrain and a small number of obstacles, with a ridge running off to the right side when positioned at the starting location and orientated towards the goal. For the other scenarios within Config_combos_0 the starting location was atop a plateau, on which additional obstacles were placed. The goal was located on a lower region past the obstacles, with a steep slope present between the goal and the obstacles. Config_combos_2 was the same as the second pair of scenarios in Config_combos_0 but with the start and finish points reversed.

Config_combos_2

Upon following trials, the various weightings of h-values were limited to 1, 1.1, 1.2, 1.4 and 2 as the variance between results from different weighting appeared

	Success	Failure		Incomplete
H-value weighting	Goal reached	No path to goal	Hazard traversed	Error
1.0	3	5	2	0
1.05	1	3	4	2
1.1	0	3	4	3
1.15	3	6	1	0
1.2	0	5	4	1
1.3	0	7	3	0
1.4	0	0	3	7
1.5	0	10	0	0
2	0	10	0	0

Table 4.3: Results from the fourth configuration within Config_combos_0.

	Success	Failure		
H-value weighting	Goal reached	Became stationary, no path	Stuck alternating between points	Hazard traversed
1.0	15	1	1	3
1.1	15	4	1	0
1.2	17	3	0	0
1.4	2	18	0	0
2.0	0	20	0	0

Table 4.4: Config_combos_2.

less significant than anticipated. Config_combos_2 was the next configuration to be used, containing one location-destination setting which was used with two different obstacle configurations. The results from Config_combos_2 were found to be more varied across weightings, but less so between the two different obstacle setups, which appear to have had limited affect, so their results were combined and are given in Table 4.4.

Trial sets A, B and C

Using Config_combos_2 again (as from the previous trials it was considered a more interesting combination), other facets of the simulation were then varied. Initially the number of steps occurring between compulsory path re-evaluations were varied. These trials were given the suffix of A0, A1 and A2 which were 2, 5 and 10 steps, respectively. Next the number of steps was reset to one and the clearance buffer the robot required around itself was increased from 100 mm to 300 mm. The trials were repeated and given the designation/label of B. Across

	Success	Failure			Incomplete
H-value weighting	Goal reached	Became stationary, no path	Stuck alternating between points	Hazard traversed	Error
A0					
1.0	7	6	0	7	0
1.1	9	8	1	2	0
1.2	13	3	0	4	0
1.4	0	20	0	0	0
2.0	0	20	0	0	0
A1					
1.0	14	2	2	2	0
1.1	11	0	0	2	7
1.2	13	2	4	1	0
1.4	0	20	0	0	0
2.0	0	20	0	0	0
A2					
1.0	9	1	5	0	5
1.1	19	0	1	0	0
1.2	3	1	0	0	16
1.4	0	20	0	0	0
2.0	3	17	0	0	0

Table 4.5: Config_combos_2A.

	Success	Failure		
H-value weighting	Goal reached	Became stationary, no path	Stuck alternating between points	Hazard traversed
1.0	12	2	2	4
1.1	17	3	0	0
1.2	14	4	1	1
1.4	1	19	0	0
2.0	0	20	0	0

Table 4.6: Config_combos_2B.

this variety of parameters it was noted, from the results shown in Tables 4.5 and 4.6, that the weightings of 1.5 and 2.0 were unsuccessful in general and as such the specific paths they had attempted were inspected.

A minor alteration was made to the behaviour of the A* algorithm when no paths were found, after the specific steps involved in the paths of set 8 showed no movement occurred/paths were attempted in almost every trial. As such, the same sets of trials for Config_combo_2B were re-simulated and produced the following results, as shown in Table 4.7, which show improvement in that for the weighting of 1.4 and 2.0 a number of successful runs occured.

	Success	Failure		
H-value weighting	Goal reached	Became stationary, no path	Stuck alternating between points	Hazard traversed
1.0	14	0	2	4
1.1	17	0	2	0
1.2	16	1	0	0
1.4	14	2	4	0
2.0	14	0	0	6

Table 4.7: Config_combos_2B re-simulated.

	Success	Failure			Incomplete
H-value weighting	Goal reached	Became stationary, no path	Stuck alternating between points	Hazard traversed	Error
1.0	1	9	2	0	8
1.1	0	0	20	0	0
1.2	5	7	2	0	6
1.4	13	2	4	0	1
2.0	3	6	4	0	7

Table 4.8: Config_combos_2C.

Another variation was tested, which was increasing the step sizes to 500 mm. This trial was labelled C and was chosen for comparison with A (specifically A1, which was 5 moves of 100 mm) to determine whether performance was better with many small steps or with fewer large movements. The results generated by these sets of trials are given in Table 4.8. Note: The results for this trial were produced on a different computer, which had become available for running some simulations, to the other pilot trials.

Following the modification of the incomplete path behaviour, the trials were also altered in the range of weightings used for testing. The original values were chosen to rise in growing increments, so as to give more coverage to closer values but also include values of much greater difference. Upon the second set of trials, where only every second weighting value was used, it was noted that the first few values followed a power of two pattern in their variance from the original weighting. It was decided to follow this pattern through and as such the final two weightings were changed from 1.5 and 2.0 to 1.6 and 1.8, respectively.

It was also found that the end condition of trials were not always being printed to file, as when exiting due to being caught alternating between two points the

relevant exit status was not being passed.

Trial sets D, E, F and H

Additional trial sets were tested which matched up with previous sets but with the basis for the network node creation modified to result in fewer nodes being produced or retained and hopefully less re-processing occurring between successive iterations. Set D is the same as the default settings but with network changes of reducing the visible distance and increasing the minimum allowable distance between nodes, being applied. Sets E and F correspond to sets A1 and C, respectively, having the same settings except for the aforementioned changes to the basis from which the network node creation occurs. Set H was related to the default setting and Set D, with the maximum distance across which nodes could be linked being reduced below the level of the visible distance. Like previous sets, these trials were tested on Config_combos_2. No issues were detected with the results, with the previous sets appearing to have covered all areas in which the trials may wish to be altered.

Config_known

Another configuration was Config_known, which provided a flat obstacle free landscape that enabled the use of pre-known/made nodes. The pre-made nodes were used to isolate the path planning section from the earlier hazard identification and node network creation stages. The separation of modules allowed comparison of performance when using the sections compared to omitting them and using idealised/pre-known node positions. These pre-made nodes could potentially be used to test the algorithms with a grid network, however in this instance they were not put to that use.

This configuration was used to visually monitor the performance of the hazard identification and the network node creation, and it was found that both tasks appeared to function as expected. Figure 4.2 contains a few screen shots taken during this testing, which demonstrate the ability to correctly set the hazard status of ter-

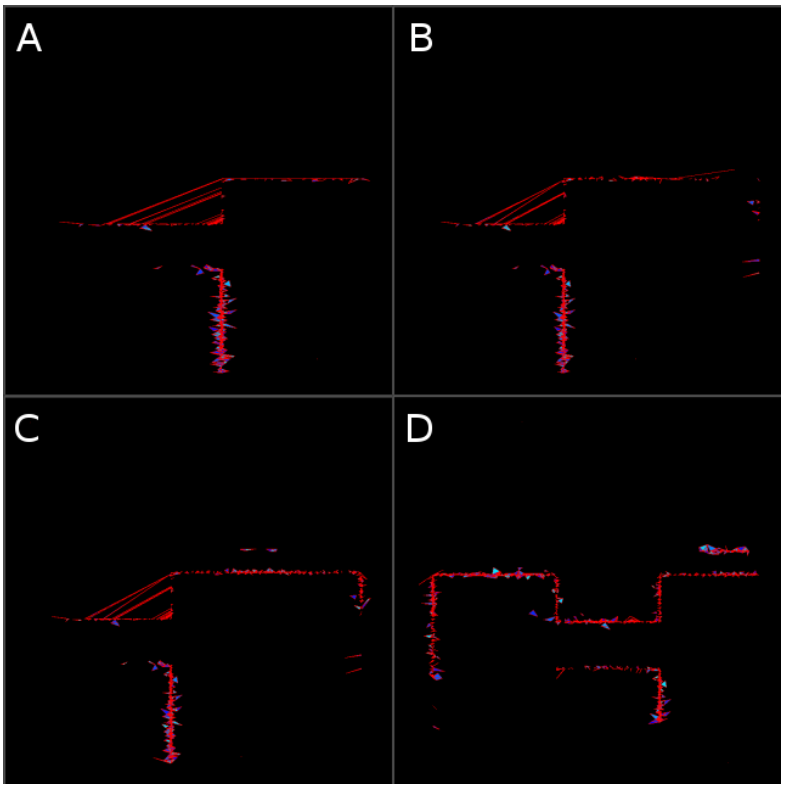

Figure 4.2: Visual depictions of nearby hazard triangles within the set identified by the robot. A $\rightarrow$ C were produced as the robot moved to the right, while D is from later on, when the robot has explored to the left of the initial position of A.

rain faces, retrieve this information and through motion inferred by sequence, the avoidance of hazardous regions. Frames A $\rightarrow$ C of Figure 4.2, follow the Robot as it explores to the right, while Frame D is from later in the simulation when it has exhausted its search of the right and has entered a region to the left to further explore potential paths. The robot's node network of potential paths was also monitored and found to have a gradual expansion as expected, as it moved into previously unknown regions. Correct linking was found to occur within current network nodes and between current nodes and the nodes from previous iterations/-positions.

Config_combos_5

One configuration which was briefly tested involved a gentle rolling terrain, with regular positioning of multiple obstacles. This involved more obstacles than had been included in other scenarios and as such lead to the discovery of a bug in the code, due to a missing break statement, which could cause objects to be written to a full array before it had been reallocated more memory. With the addition of the omitted break statement, the bug was fixed as one of the final acts of the pilot testing session, before moving on to gathered experimental results.

4.5.2 LPA* algorithm

Due to an intermittent bug existing somewhere within the memory management of the LPA* algorithm, trials using it had each scenario run as two lots of five rather than one set of ten, so as to avoid the risk of the bug being encountered and hence the simulation exiting before completing the whole set.

4.5.3 Miscellaneous Configurations

Some other configurations which were tested but not chosen for extended testing were Config_combos_3 and Config_combos_4, which were flat terrains on which a single large maze obstacle and a larger alternate maze obstacle, were each placed. The mazes had been created using Blender 3D (a modelling suite), and exported as .3ds objects, however there were some issues with adding them and their meshes to the simulation terrain. Subsequently the configurations were omitted as they could not be relied on to run correctly.

Chapter 5

Path Planning Results

After the pilot tests were completed, full experimental testing was undertaken with the number of repetitions for each configuration pair within a config_combo being extended to 50, producing 100 trials for each weighting.

The initial configuration file of Config_combos_0, consisted of four scenarios, produced through combining two terrain settings and two obstacle settings. Due to the similarities between the results for the first pair of scenarios (having the same terrain file but differing obstacle files) and between the results of the second pair of scenarios, but differences between the pairs, it was decided to separate the second pair out of Config_combos_0 and into another configuration file, so as to better group the outputted results. This decision was made part way through the generation of experimental results while beginning to calculate and plot the statistics for some of the trial sets, as such for some trials the data was outputted into a combined file. For these combined results only the number of occurrences of different end conditions can be provided for Config_combos_0 and Config_combos_1 (extracted from SimResults.txt) as filtering the data within the Actual_Steps.txt file is too time consuming and difficult to ensure it is done correctly.

Of primary interest within the results from these simulations, are the mixture of completion status, the average number of movements occurring for successful completions and also the general nature or pattern of paths when overlaid.

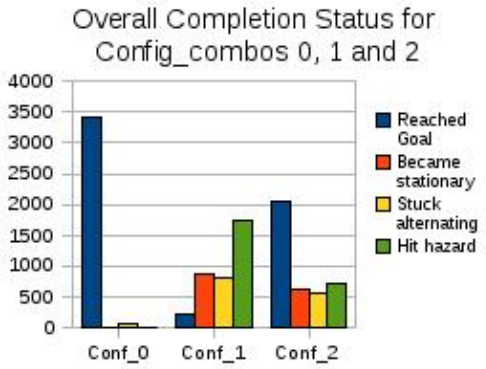

Figure 5.1: Comparison of the summed/generalised results produced using each configuration file.

5.1 Config_combos 0, 1 and 2

A general idea of the complexity of each of the scenarios is demonstrated in Figure 5.1, which gives the completion status for the results generated using each configuration file, combined together according to the respective file used. Config_combos_0 can be seen to be the least difficult configuration, through the clear predominance of successful completions as the most frequent end condition. Directly contrasting the success rate of Config_combos_0, Config_combos_1 is shown to have a very low occurrence of success, with the three alternative reasons for completing a simulation all occurring many times.

5.2 Algorithms

5.2.1 A* Algorithm

An observation about the simulation runs for the default parameter setting and A* algorithm, was that based on external factors it would appear they potentially ran slower than some of the modified sets. This observation was made due to the local temporary directory in which data were recorded, clearing some of the earlier files which it is expected was due to age and inactivity, possibly in combination

with scheduled cleans. The simulations were run from the local /tmp/ directory of each computer, with data temporarily recorded there in order to avoid excessive read/write operations on the home directory as it was a network drive. Upon completion, the shell file which had called the Simulations proceeded to transfer the results on to the network drive, however this occurred too long after the earliest data were recorded and trial sets 0 for all three configurations were lost and required to be re-simulated. These losses were an oversight, with research into the exact basis the computers were using for clearing the tmp directory needing to have been investigated.

Comparing the results of the A* algorithm under default settings and across all three configuration files, as shown in Figure 5.2, configuration 0 and 1 produce results which are polar opposites, respectively being highly successful and very poor. The results generated for Config_combos_2 provide a mixture of completion status but are distinctly closer to Config_combos_0 than to Config_combos_1. In terms of steps necessary to conclude a trial, Figure 5.2 indicates that trial runs on Config_combos_2 consisted of a much higher number of movements. While the greatest straight line distance between the start position of the robot and the goal location is for Config_combos_0, the other two configurations require greater deviations to avoid regions which are not traversable and hence will generally consist of more movements.

5.2.2 LPA* Algorithm

Because of time constraints and in light of earlier results, it was decided to limit the experimental trials of the LPA algorithm to configuration files zero and two, and reduce the number of h-value weightings tested to three (1.0, 1.2 and 1.8). The simulations of the LPA* trial sets occurred after completing all the initial simulations for the A* trial sets. At this point the computers began to perform poorly, with the linux graphical desktop and the file manager becoming slower to respond, in some cases to the point of requiring logging out and in again or a reboot. As the current installation of Fedora on these computers had not been used before (being the under graduate computer lab), it is not clear whether this

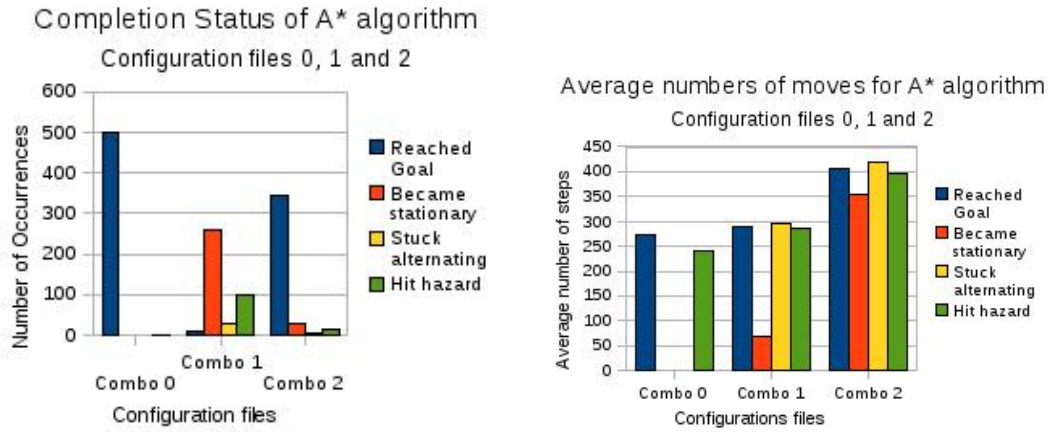

Figure 5.2: Left: Completion status for A* algorithm, on Config_combos_0, Config_combos_1 and Config_combos_2. Right: The average number of movements occurring for each completion status.

was a side-effect of prolonged usage, multiple log in sessions of the same user or the simulation tool. However, as the network administrator had mentioned that there were issues with the installation in terms of the graphical desktop and the file manager, plus regular system messages informing that a package dependency could not be resolved, it is felt the problems are fully or at least significantly due to issues external to the simulator.

The LPA* algorithm performed well on Config_combos_0 with a high number of successful completions, while in contrast the results produced with Config_combos_2, given in Figure 5.3, having few successes and instead have the robot becoming stationary most of the time.

5.2.3 State-based Algorithm

The simple state-based algorithm was briefly tested upon Config_combos_0 and Config_combos_2, with very poor results. It was found that, as shown in Figure 5.4, three paths were followed for Config_combos_2, of which two were almost identical save for a slight deviation, while the third path ran quite closely to the other two. With about 70 moves on average, all three of these paths were fairly

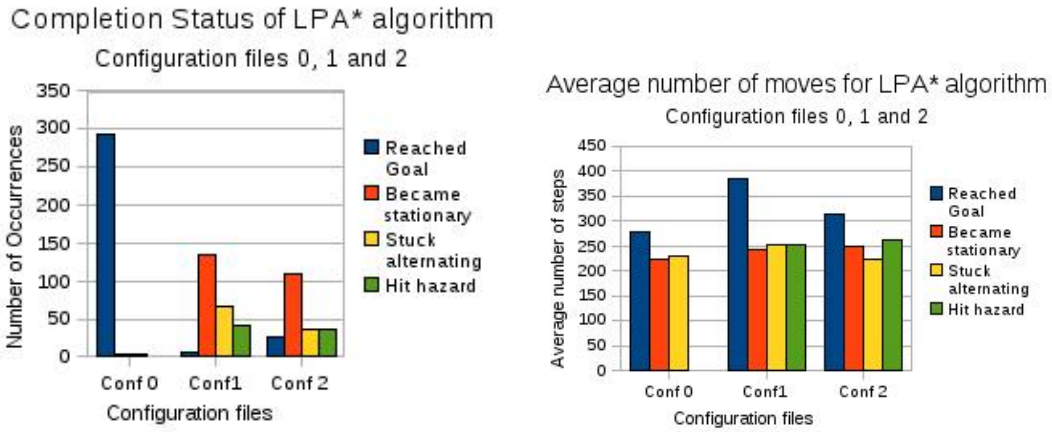

Figure 5.3: Left: Completion status for LPA* algorithm, on Config_combos_0 and Config_combos_2. Right: The average number of movements occurring for each completion status.

short in comparison to the paths generated by the other algorithms which even in cases of not finding the goal, would travel upwards of 150 steps on average. For Config_combos_0 the results were somewhat better, with the robot reaching the goal on nine out of forty seven occasions, which can be seen in the overlay of paths, as shown in the bottom right panel of Figure 5.5.

These poor results are likely due to the rushed conception and design of the algorithm with no performance based testing, and are unlikely to reflect the general performance of other case-based algorithms. However, the fact that in the simpler scenarios provided by Config_combos_0, this quick and simple design was capable of inter-operating with the other modules of the simulator to avoid an obstacle and reach a path, is testimony to how the preparation of data and creation of node network is a significant part of path navigation.

5.2.4 Comparison of Algorithms

Over the regions the three configuration files covered, the A* and LPA* algorithms were respectively run with 5 and 3 values of h-value weighting. The State-based algorithm had no equivalent variable for which the value could be weighted. As

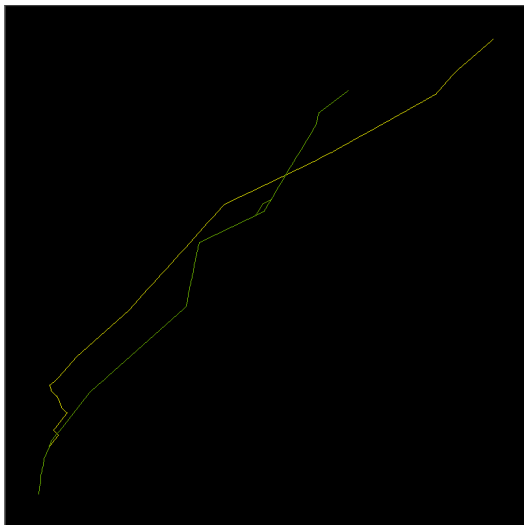

Figure 5.4: Overlaid paths generated by a simple state-based obstacle avoider for Config_combos_2.

seen in Figure 5.5, the A* and LPA* algorithms produced very similar paths when run on Config_combos_0, which is not surprising. For the LPA* results, the paths appear to be grouped slightly tighter but this is most likely due to the quantity of trials being lower, with the A* algorithm having been run with 5 weightings compared to LPA* being trialled with only 3. The results from the State-based algorithm were quite different to those of the other two algorithms. However, there is a region in the centre of all the State-based results which appears to be similar in part, to the results produced by the A* and LPA* algorithms. This region of similarity does not continue on for the State-based algorithm, as it did with the other algorithms, as it would appear the state-based algorithm failed to negotiate getting past the obstacle.

For Config_combos_1 the sets of paths produced by the A* algorithms, presented in Figure 5.6, were not as tightly bunched as had been in seen with Config_combos_0. The presence of two regions containing hazardous obstacles is quite clearly defined in the overlaid paths, with the absence of paths due to skirting the areas. Looking at Figure 5.7, the results for the A* algorithm are a combination of some tightly grouped paths reaching the goal and some other paths

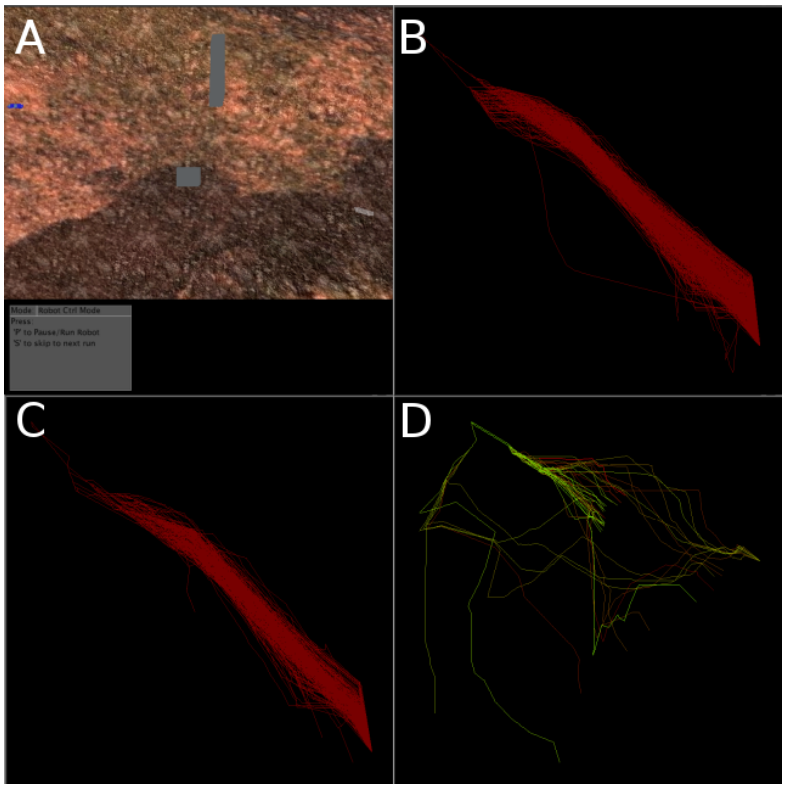

Figure 5.5: Config_combos_0 terrain overview plus overlaid paths generated by the A*, LPA* and State-based algorithms, given as B, C and D, respectively.

which spread out in a diffuse manner as they search for a suitable path to the goal, while with the LPA* algorithm all the path lead directly to the goal with very little divergence from the core path.

5.3 H-value weightings

5.3.1 A* Algorithm

All trials involving the A* algorithm were run under the 5 different h-value weightings of 1.0, 1.1, 1.2, 1.4, and 1.8. Looking at the results from individual weightings there is almost no variation between the number of occurrences of the different completion status, as is clearly visible in Figures 5.8, 5.9 and 5.10. The av-

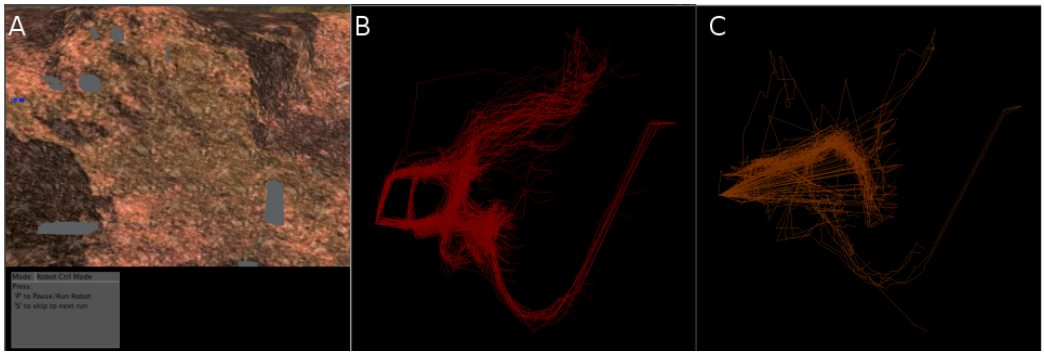

Figure 5.6: Config_combos_1 terrain overview plus overlaid paths generated by A* algorithm, given as B and C, respectively.

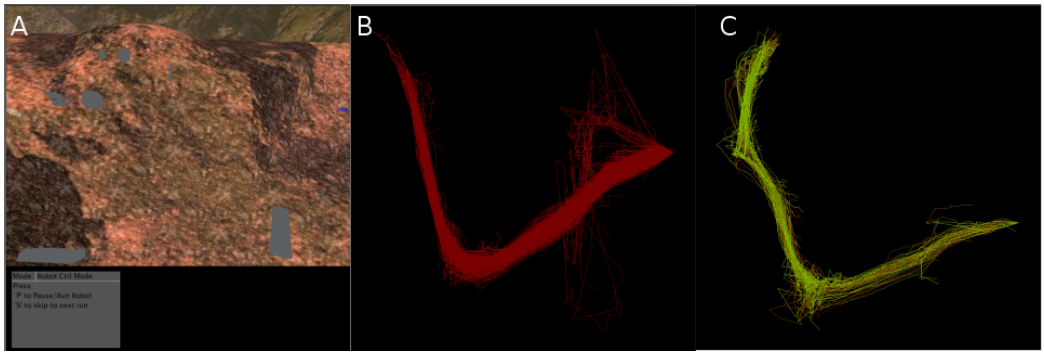

Figure 5.7: Config_combos_2 terrain overview plus overlaid paths generated by A* and LPA* algorithms, given as B and C, respectively.

erage number of steps taken before the trials exited were also very similar across the different weightings, with the only notable difference being for weightings of 1.2 and 1.4 in Figure 5.9. The results indicate that the goal was reach after a much greater number of steps for the h-value weightings of 1.2 and 1.4, in comparison to the successful cases of a weighting of 1.8 and all the other cases of weightings 1.2 and 1.4, but as there were very few occurrences no conclusion had be drawn with confidence about the successful paths under those weightings. With Config_combos_2 the average number of steps taken for unsuccessful completions appears to decrease as the h-value weighting increases, which potentially suggests that weighting the h-value is a hindrance but once again the number of occurrences is too low to add any confidence to perceived trends.

These other trials also produced results which showed no notable difference between the specific results from the different values of the h-value weighting. Therefore, upon further testing the weightings were grouped together and presented as a single set due to the similarity across all other trials.

For the two scenarios within Config_combos_1 the results were quite different in that for the first one the same path was chosen each time and failure occurred at the same point, becoming stationary due to being unable to determine/detect any paths. As such, the number of occurrences for the status of becoming stationary is very high, which when comparing Figure 5.9 to the results produced for Config_combos_1 by others sets, gives a distinctly different result to the trend of completion status whereby hitting a hazard is the most common occurrences for this configuration. As the second scenario, which involved the same region of terrain but with different obstacles placed within it, produced results akin to the other trials it must be surmised that this early halting was specific to the scenario rather than being an overriding behaviour.

Note: The results for the h-value weighting of 1.0 were lost for the A* algorithm as previously mentioned but unfortunately re-simulations were not completed in time for inclusion.

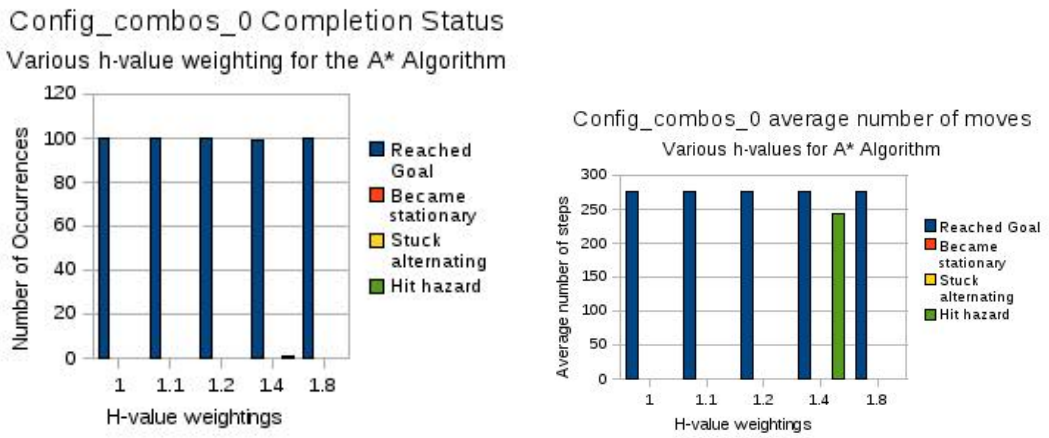

Figure 5.8: Left: Completion status of various h-value weightings for A* algorithm, on Config_combos_0. Right: The average number of movements occurring for each completion status.

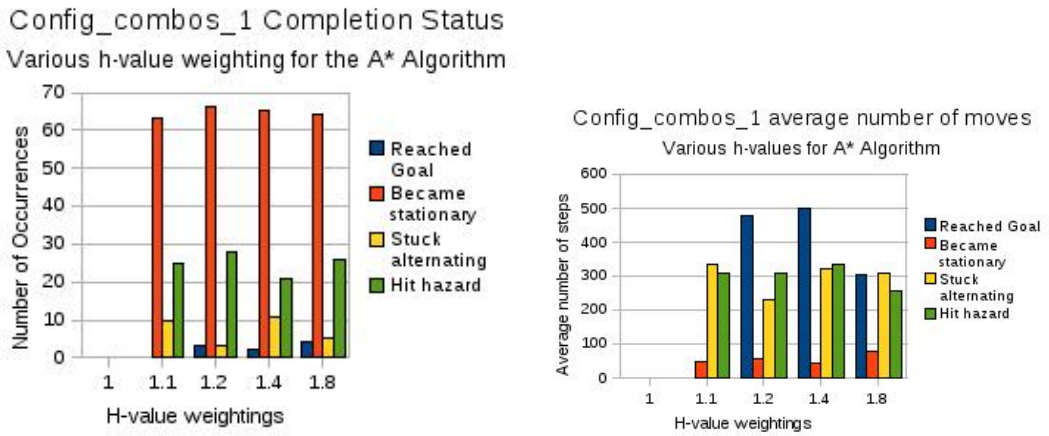

Figure 5.9: Left: Completion status of various h-value weightings for the A* and LPA* algorithms, on Config_combos_1. Right: The average number of movements occurring for each completion status.

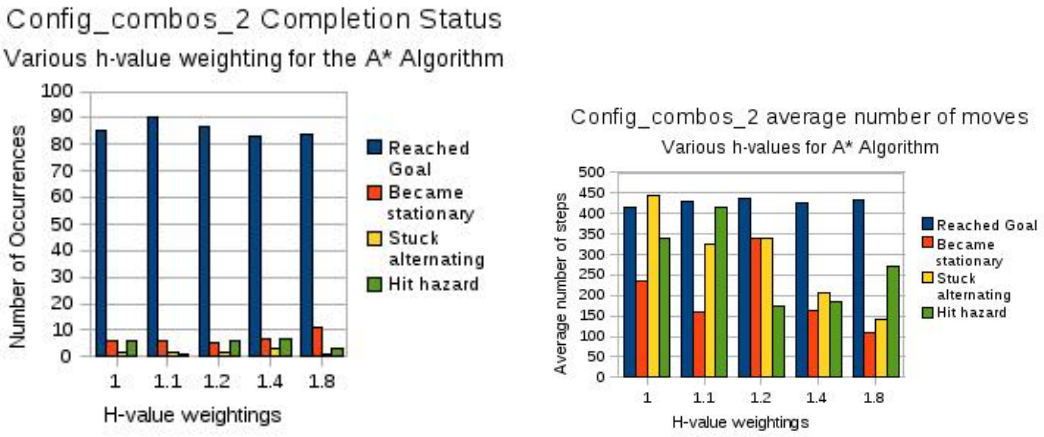

Figure 5.10: Left: Completion status of various h-value weightings for A* algorithm, on Config_combos_2. Right: The average number of movements occurring for each completion status.

5.3.2 LPA* Algorithm

As mentioned previously, the trial sets for the LPA* algorithm involved three different h-value weightings of 1.0, 1.2, and 1.8, and were trialled on Config_combos 0 and 2. The results from the trials when the h-value weighting was 1.0 and Config_combos 2 was used, were lost when they did not get copied in to results directory and were left on the /tmp drive, where they were later deleted. This loss is somewhat significant as fewer weightings had been trialled, however from Figures 5.11, 5.12 and 5.13, as well as the initial pilot trials, it can be seen that the results for the LPA* algorithm were like those of A*, in that the various h-value weightings did not appear to make a significant difference to the planned paths. The results gathered using the LPA* algorithm contained no surprises, with Figures 5.11, 5.12 and 5.13 all being similar to the corresponding ones of the A* algorithm, which suggests that the LPA* algorithm was implemented correctly and without behavioural bugs.

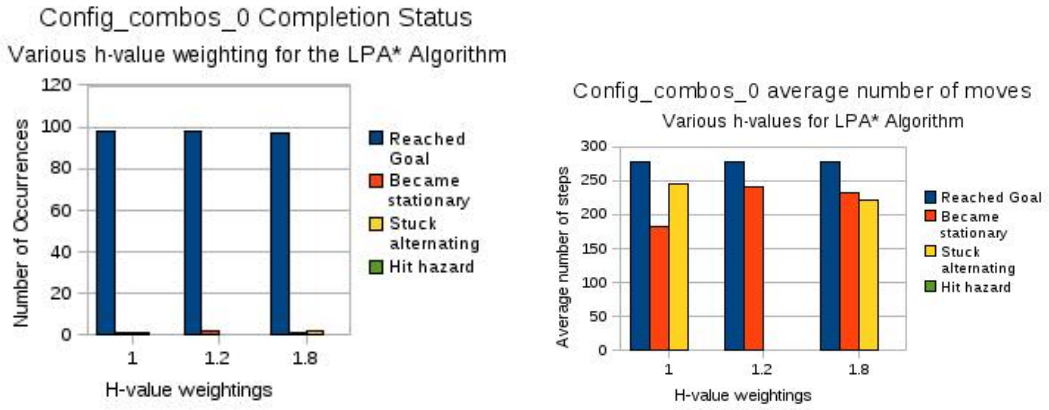

Figure 5.11: Left: Completion status of various h-value weightings for the LPA* algorithm, on Config_combos_0. Right: The average number of movements occurring for each completion status.

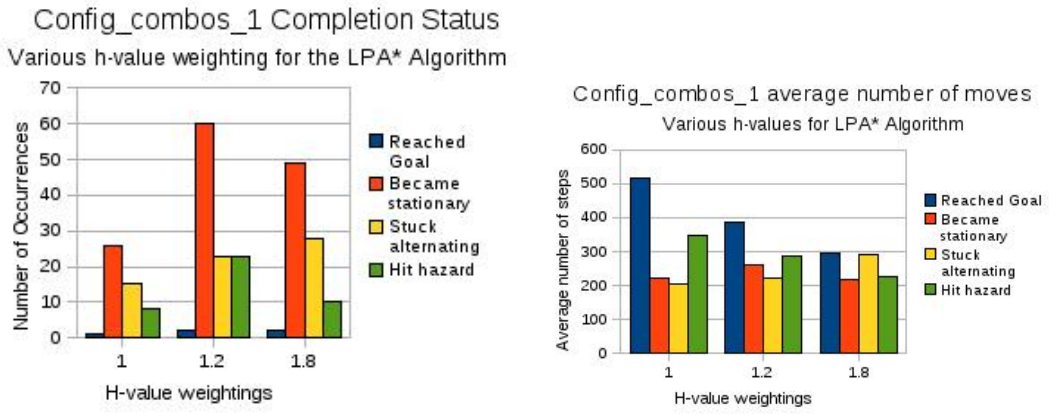

Figure 5.12: Left: Completion status of various h-value weightings for the LPA* algorithm, on Config_combos_1. Right: The average number of movements occurring for each completion status.

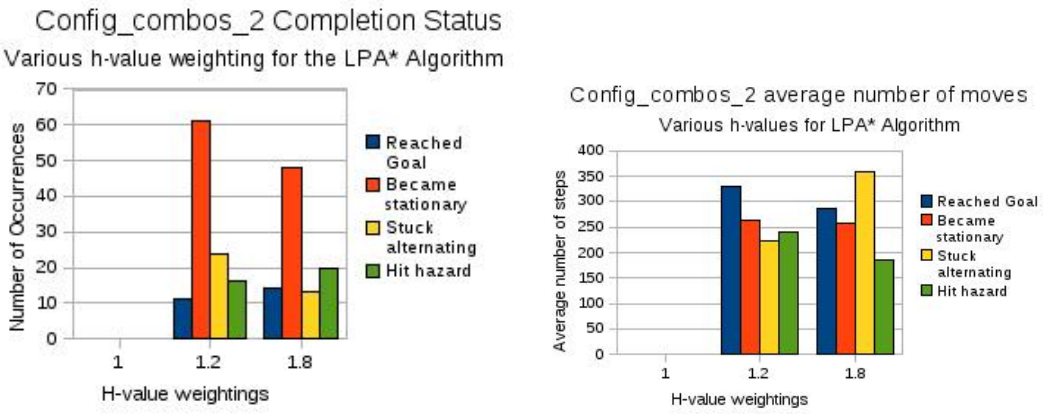

Figure 5.13: Left: Completion status of various h-value weightings for the LPA* algorithm, on Config_combos_2. Right: The average number of movements occurring for each completion status.

5.4 A-Series Trial Sets

Figure 5.14 contains the results across sets A0, A1 and A2, from Config_combos_0, with Figure 5.15 and Figure 5.16 showing the results from Config_combos_1 and Config_combos_2, respectively. One might note that the total number of completion results in Figure 5.14 is noticeably lower than in other cases, the reason is that the results of only one of the two scenarios in Config_combos_0 were gathered for each weighting of each set. The configuration file was accidentally edited during simulation and so for some weightings the first scenario was skipped. As the results displayed are the combination across all weightings, it was felt it was necessary to endeavour to have each weighting have a reasonably equal influence and thus omit the relevant data from those sets which had fully completed the scenarios of Config_combos_0. For the completion status which had a single line of data for each trial run removing the data was simple. However, the data for the sequence of steps taken for each trial run was significantly greater and more difficult to filter, which resulted in the re-inclusion of all data for Figure 5.14. For Figures 5.17, 5.18 and 5.19, it had been decided that all of the available paths were overlaid since the visual representation of the paths is more general and less definite

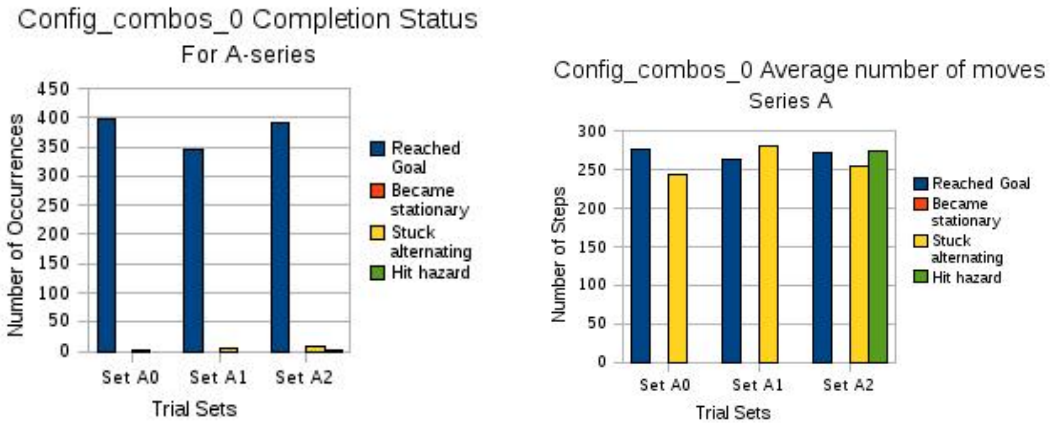

Figure 5.14: Left: Completion status for the A-series of alterations with the A* algorithm, on Config_combos_0. Right: The average number of movements occurring for each completion status.

than the statistical characteristics being shown in the charts, and this decision did not change. Across all three sets of trials, the results from the A-series produced paths which were very alike, as Figures 5.17, 5.18 and 5.19 show.

Once again the results for Config_combos_0, have resulted almost solely in successful runs being completed, as Figure 5.14 demonstrates. While the results shown in Figure 5.15 for Config_combos_1 are not particularly good, they do have a greater presence of successful runs and a lower incidence of encountering hazards than can be seen in Figure 5.21, for Sets D, E and H, which are presented later. Interestingly it would appear that each of the sets within the A-series managed to find a path down the slope located directly between start and finish, which did not exceed the maximum slope angle to which the robot was limited. This could be due to an error in the simulator's detection of hazardous traversals, however as the path appears in the same spot across all the sets this suggests it is an actual path, though it does not rule out it being a random error.

Note: The display/recreation of paths appears to show that the robot has travelled past the goals. This was originally thought to be due to not taking into account the affect of uphill/downhill travel, whereby the the absolute distance covered is actually further due to the z-component, while the recreation displays

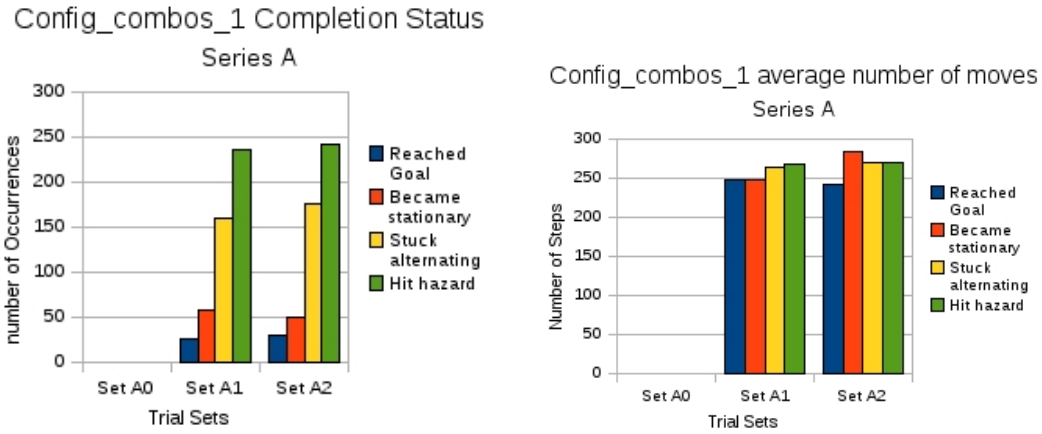

Figure 5.15: Left: Completion status for the A-series of alterations with the A* algorithm, on Config_combos_1. Right: The average number of movements occurring for each completion status.

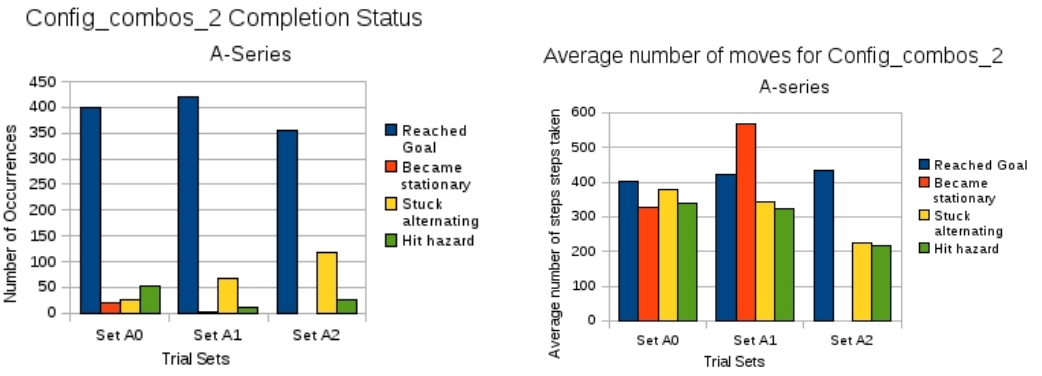

Figure 5.16: Left: Completion status for the A-series of alterations with the A* algorithm, on Config_combos_2. Right: The average number of movements occurring for each completion status.

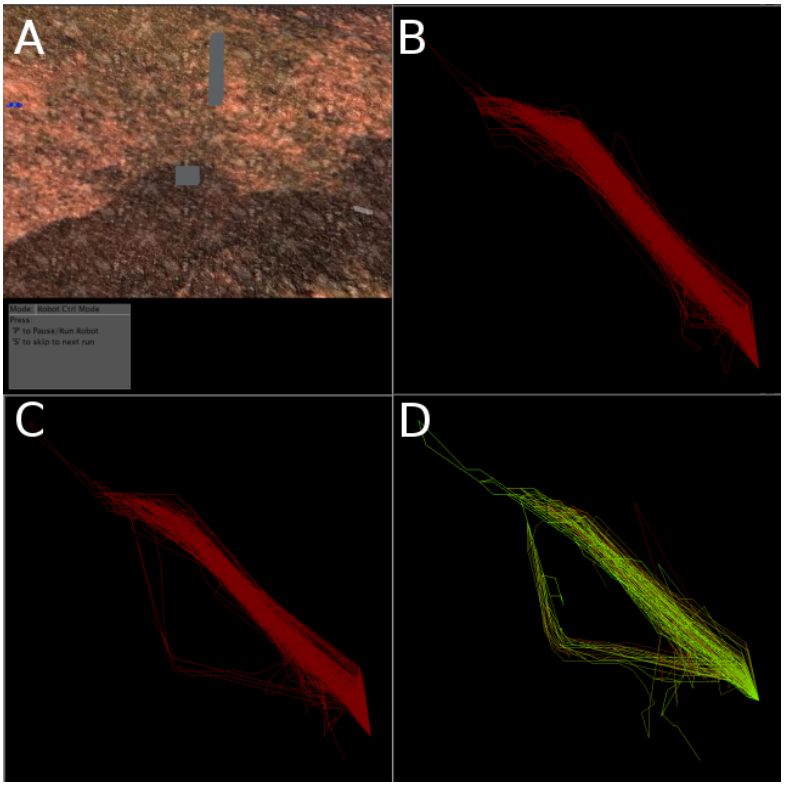

Figure 5.17: Overlaid paths generated by the A-series of variation, with the A* Algorithm on Config_combos_0. (A) Terrain overview (B) Set A0 paths, (C) Set A1 paths and (D) Set A2 paths.

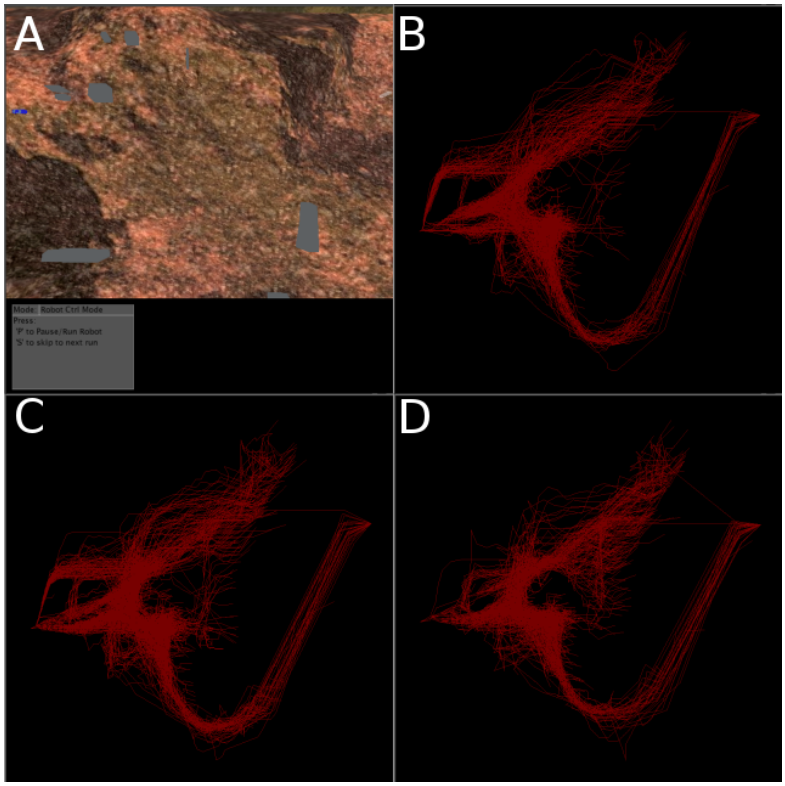

Figure 5.18: Overlaid paths generated by A-series variation with A* Algorithm on Config_combos_1. (A) Terrain overview (B) Set A0 paths, (C) Set A1 paths and (D) Set A2 paths.

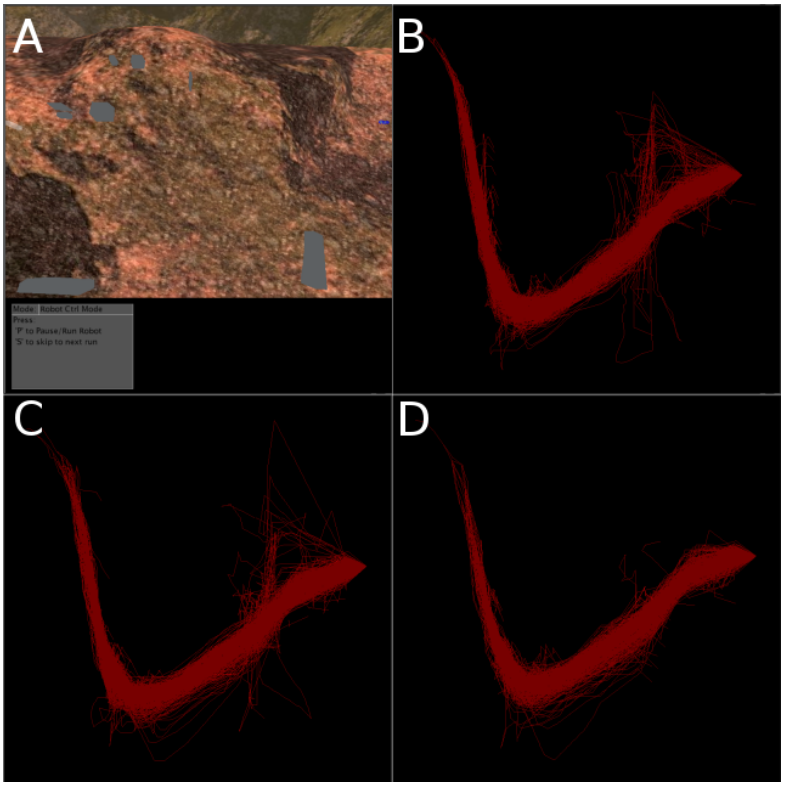

Figure 5.19: Overlaid paths generated by A-series variation with A* Algorithm on Config_combos_2. (A) Terrain overview (B) Set A0 paths, (C) Set A1 paths and (D) Set A2 paths.

the movements as if derived purely from the XY-plane. Considering the paths of both Config_combos 1 and 2, the same overshoot appears to occur for both in spite of moving in opposite directions. As such, it may be that the display is twisted relative to the orientation at which it is believed to be presented.

5.5 Trial Sets D, E ,H

Due to a numerical overflow occurring during the calculation of distances between points, for trial sets D E and H, the Config_combos_1 configuration file exited with an error. This was not detected during pilot testing, as not all potential combinations of trial sets and configuration files were tested, the aim having only been to ensure each trial set or configuration file was tested at least once. This bug was fixed through changing the data types used within the function. The original data type used were floats (floating point precision) and this was changed to using doubles (double floating point precision) to avoid the numerical overflow. With the bug fixed, the sets where then completed without further problems.

For Config_combos_0 the three sets of D, E and H, which all have the visible distances halved and the minimal distance between mapNodes doubled, prove to be in line with other trials set run on Config_combos_0 in having very high rates of success as can be seen in Figure 5.20. Of the instances in which the simulation did not end upon the successful arrival at the goal, the outcome was limited to not becoming stationary due to not finding further paths or due to getting stuck in a cycle of alternating between two positions, with no trials hitting any hazards. Figure 5.20 also shows that the number of steps taken in reaching the goal was homogeneous across the sets. It may be noted that Set H has a lower occurrence (but similar success rate) than the other two sets. This is because the simulation did not quite complete the full 500 trial runs, encountering a segmentation fault when creating and loading a new simulation environment. A bug/fault believed to be located in the code for the Irrlicht graphics engine as opposed to the code written for this project.

As with other trial sets, the results for D, E and H on Config_combos_1 have

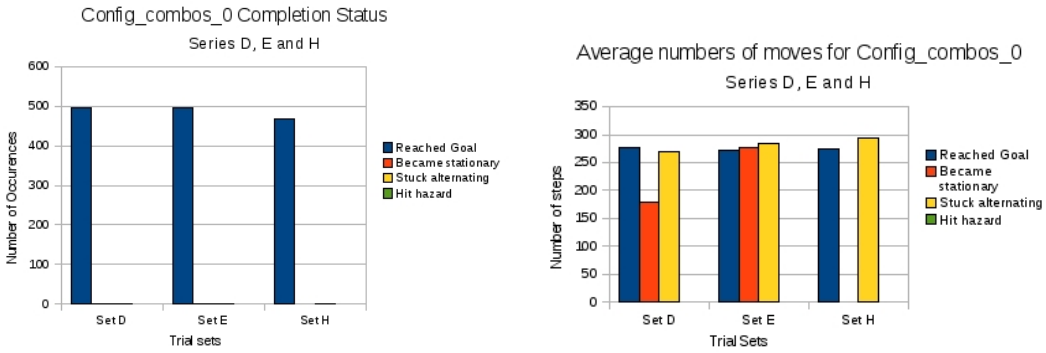

Figure 5.20: Left: Completion status for sets D, E and H with the A* algorithm, on Config_combos_0. Right: The average number of movements occurring for each completion status.

a high failure rate consisting predominately of encountering hazards. The occurrence rates for the other non-successful status vary between the three sets, as evident in Figure 5.21. Sets D and E have a similar numbers of occurrences for the second and third most common status. However, which status is the second and third most common, is reversed between the two, with becoming stationary being the more common of the two while for set E it is being stuck alternating. Though the number/sample of successful runs is very low, it is still notable that the average number of steps taken in reaching the goal is much higher for set E than the results of any other set running on Config_combos_1. It is debatable whether this is a good result, indicating set E may find a path even in difficult circumstances, or whether it indicates that the paths chosen by set E are less efficient than the others. More testing across a variety of scenarios is necessary to make informed conclusions about the trial sets. The number of movements before trials terminated unsuccessfully, was reasonably similar both between the status and between the sets, though in each case set H appears to have encountered trouble earlier (within fewer moves than the others). Looking at Figure 5.24 it looks like the cause of the lower average number of steps before termination may be due to a greater tendency to attempt to traverse to the left which featured more obstacles.

As expected of Config_combos_2, the completion status across the trial sets

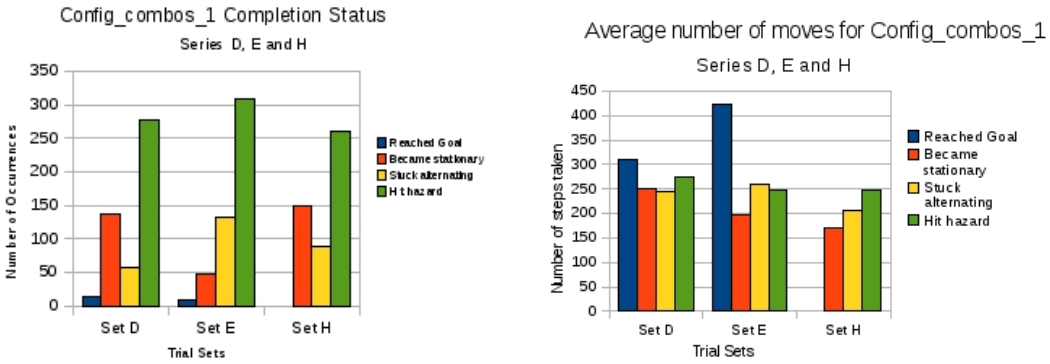

Figure 5.21: Left: Completion status for sets D, E and H with the A* algorithm, on Config_combos_1. Right: The average number of movements occurring for each completion status.

was a lot better than with Config_combos_1. Looking at Figure 5.22 it appears that Set H did not perform as well as the others, with half as many successes as Set D and having the highest number of instances of hitting a hazard. Looking back upon Figures 5.20 and 5.20, it would appear that Set H has consistently performed the worst of these three variations based on reducing the resource requirements of the robot. With Set H having the distances across which nodes may be linked reduced further than the other two sets it had been expected the results would show greater performance due to a more simplified network of nodes being created with fewer links, however it would seem this instead handicapped the robot by reducing the options available to it for path planning. When looking at the paths produced by Sets D, E and H, shown in Figures 5.23, 5.24 and 5.25, the paths for Set H do not appear very dissimilar to the other two sets, in particular being quite like Set D in Figures 5.23 and 5.25, with the presence and spread of diffuse paths away from the most common channel of traverse.

Comparing the results of set D against the default settings to see the affect of reducing the number of nodes present in the potential path network, the first configuration file shows no difference, however looking at Figures 5.21 and 5.2, there is a difference in which completion status is the most common. For the D set of trials, the robot encountered a hazard more frequently while with the

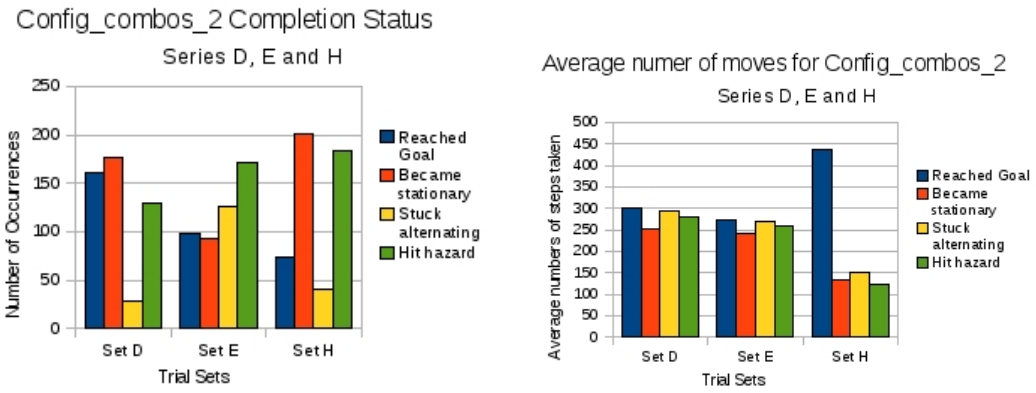

Figure 5.22: Left: Completion status for sets D, E and H with the A* algorithm, on Config_combos_2. Right: The average number of movements occurring for each completion status.

default settings the robot had otherwise been exiting when it became stationary after an average of 70 moves and could not determine a new path. The most notable difference produced by the reduced node network is seen when the results show in Figures 5.22 and **??** are compared. The success rate of the robot was drastically diminished when using the reduced network, with the robot becoming stationary or hitting a hazard on a similar number of occasions to when it reached the goal. When sets A1 and E are compared, the affect of the reduced network is very similar, with Config_combos_0 and Config_combos_1 having very minor differences while the results of Config_combos_2, as shown in Figures 5.16 and 5.22, show a significant change from a high rate of success to the other completion status become more dominant. The affect of the reduced node network is likely to be due to having fewer path options and hence with less information the robot is making poorer choices. Some method of reducing the node network selectively, to try and maintain the quality of decisions while reducing the resources required, would be beneficial.

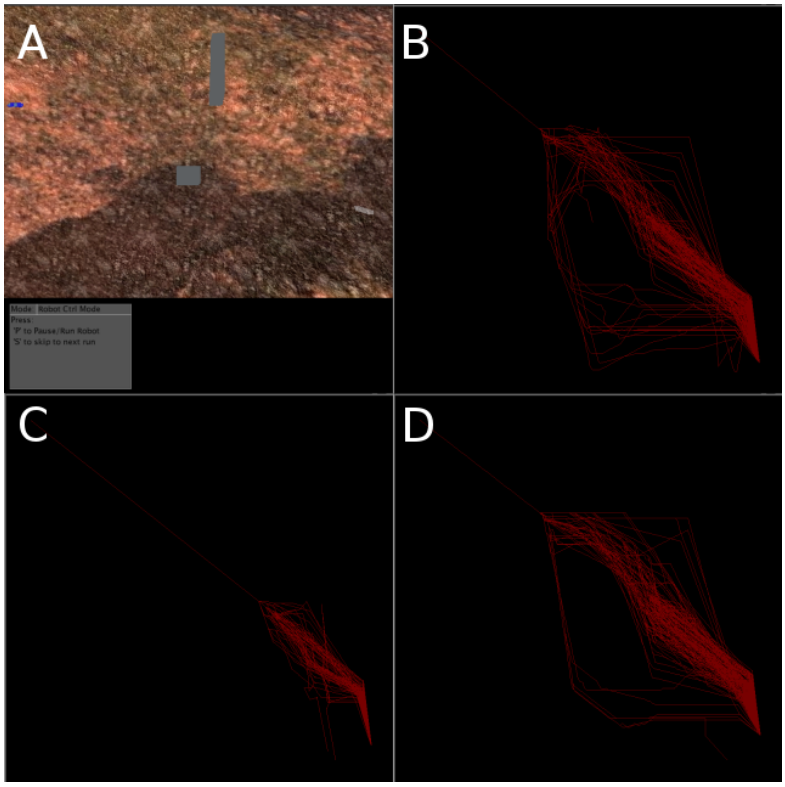

Figure 5.23: Overlaid paths generated by A* Algorithm in trial sets D, E and H, for Config_combos_0. (A) Terrain overview, (B) Set D paths, (C) Set E paths and (D) Set H paths.

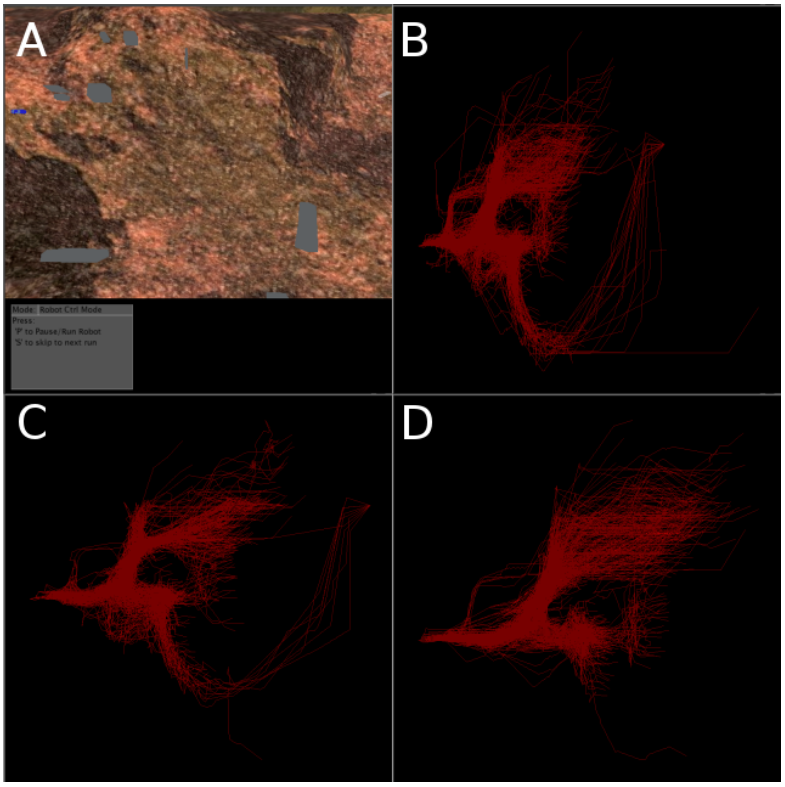

Figure 5.24: Overlaid paths generated by A* Algorithm in trial sets D, E and H, for Config_combos_1. (A) Terrain overview, (B) Set D paths, (C) Set E paths and (D) Set H paths.

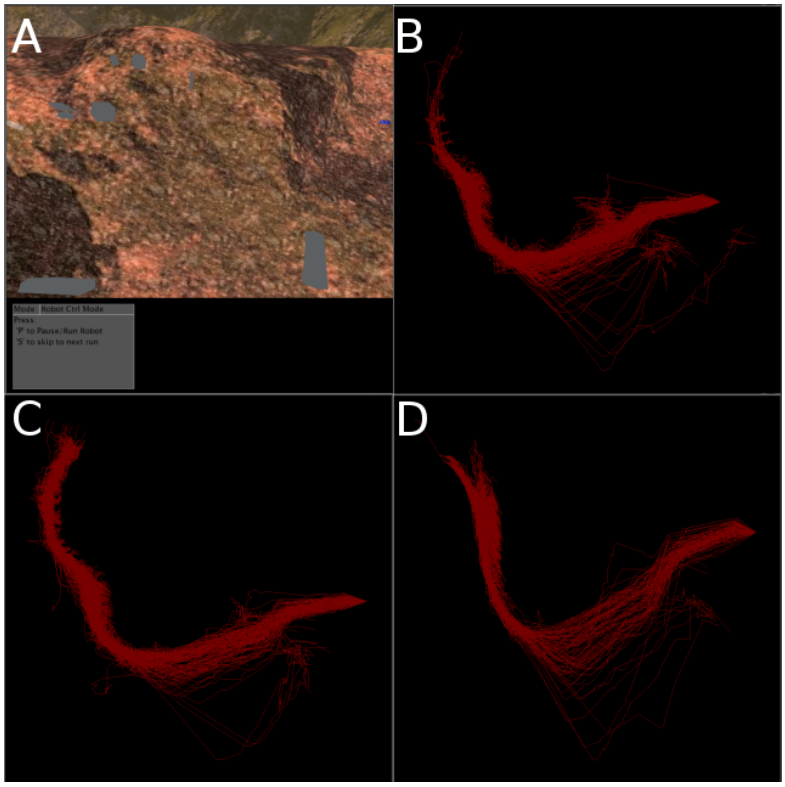

Figure 5.25: Overlaid paths generated by A* Algorithm in trial sets D, E and H, for Config_combos_2. (A) Terrain overview, (B) Set D paths, (C) Set E paths and (D) Set H paths.

5.6 Parameter comparisons

5.6.1 Trial sets A1 vs. C

The success rates of trial sets A1 and C, as shown in Figures 5.26, 5.27 and 5.28, are quite interesting to compare. For set A1 the allowable maximum number of steps between re-analysis was five, with the step sizes being 100mm, while set C consisted of movements of 500mm but with re-analysis occurring after each step. For the relatively simple scenarios of Config_combos_0 both sets produce a high number of successful traversals, though when adjusted to account for the step size of set C being five times that of set A1, it can be seen that the average number required to reach the goal is higher for set C. While set C makes less moves, if adjusted to account for the difference in distance covered by the steps of set A1 and C, set C could be considered to have travelled further in its trials which were unsuccessful. However, it should be taken into consideration that as the decision to exit due to alternating between two points, requires that a set number of consecutive iterations occur without finding a new path, the average number of steps given in the adjusted set C is slightly off/misleading.

In Config_combos_1, set C appears to fare better than set A1, with the number of occurrences for each status being closer than those of set A1 which differ dramatically with very few successes and a great number of instances of encountering a hazard. The average number of steps taken for each category of completion status, are fairly similar within each set, while set C once again appears to have moved further when adjusted to reflect step size.

Figure 5.28 shows that the results for Config_combos_2, are a reversal from the trend of set A1 producing paths which finished in fewer movements than the adjusted set C values. Set A1 has the average number of steps taken for each completion status being higher than those for the adjusted Set C.

It is also notable that set C differs from the general trend across all sets. For the other sets the results of Config_combos_2 show a greater rate of success, while set C has there being a more balanced occurrence of all the completion status. Looking at the completion status of Config_combos_1, it appears that larger steps

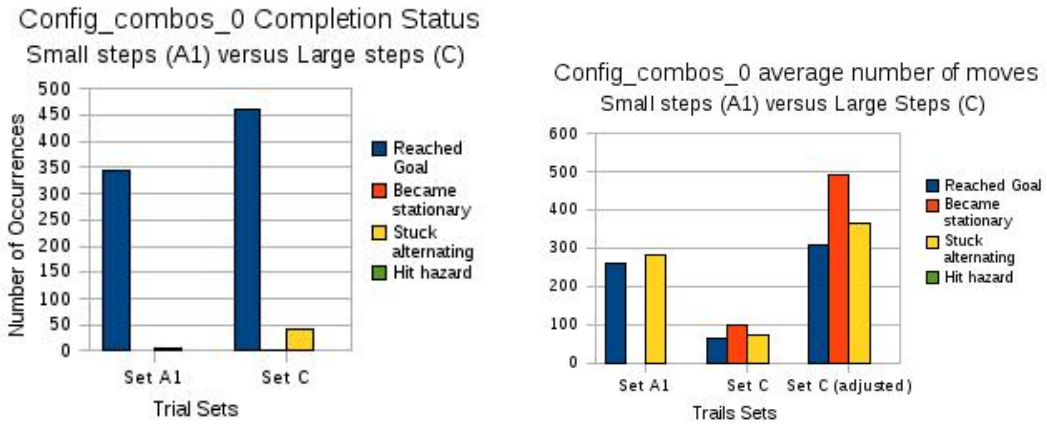

Figure 5.26: Left: Completion status for sets with small step sizes (set A1) and larger step sizes (set C), using the A* algorithm on Config_combos_0. Right: The average number of movements occurring for each completion status of the sets A1 and C plus a set adjusted set to account for the movements of set C being 5 times larger than A1.

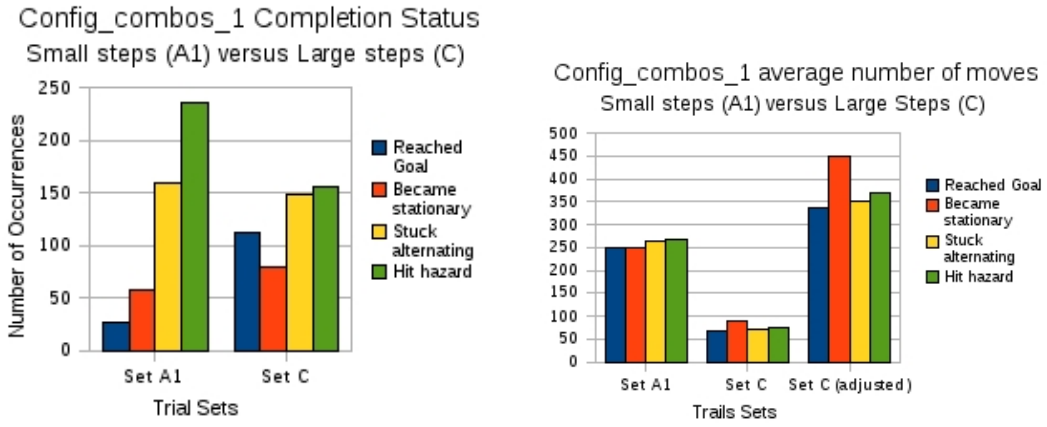

Figure 5.27: Left: Completion status for sets with small step sizes (A1) and larger step sizes (C), using the A* algorithm on Config_combos_1. Right: The average number of movements occurring for each completion status of the sets A1 and C plus a set adjusted set to account for the movements of set C being 5 times larger than A1.

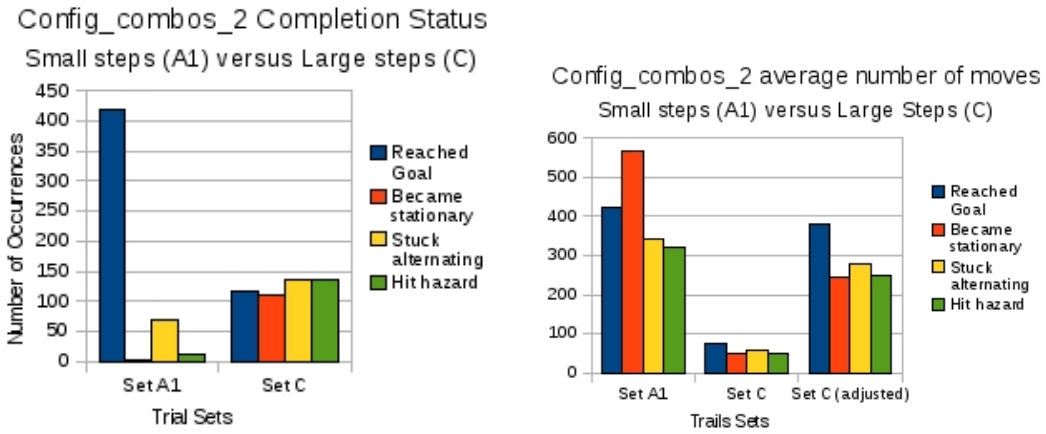

Figure 5.28: Left: Completion status for sets with small step sizes (A1) and larger step sizes (C), using the A* algorithm on Config_combos_2. Right: The average number of movements occurring for each completion status of the sets A1 and C plus a set adjusted set to account for the movements of set C being 5 times larger than A1.

sizes potentially have the effect of homogenising results, a beneficial characteristic for difficult terrains which would otherwise have few success, but less than desirable in other cases.

5.7 Noise

Initially the noise levels were set to have maximum levels of 5, 10 and 20%. However, with relation to the maximum distances it was being applied to, this corresponded respectively to errors being up to 0.5, 1 or 2m. With these high values of noise the robot was very unsuccessful and averaged 6 movements on configuration file 2 before it was no longer able to determine a path to follow. Taking in to consideration the subsequent absolute values of the errors, the MAX_NOISE parameter was decreased to values of 0.5 and 1%, to give a more realistic range of errors for the LIDAR sensing being simulated.

When the LPA* algorithm was accidentally run with noise present (at the decreased levels) the success rate was 0%. Generally, the robot became stationary

within ten moves, however on a few instances the robot managed to progress further. For these prolonged cases, the robot achieved median distances of about 40 moves before hitting a hazard.

The level of noise was in a potentially realistic range, however the distribution/frequency of it was not particularly realistic due to being based on the rand() function which gives an equal likelihood of occurrence for any value within the range. It had been hoped that this unrealistic and more extreme noise introduction may still have been manageable to some degree due to the tolerances and buffers within the design, however the potential for the noise to make a flat plane appear highly rugged and hazardous appears to have been too great. A better model for generating and introducing noise is necessary to examine the limits and performance of the path planning system in a realistic non-idealised environment.

Chapter 6

Discussion

Through the design process of this simulation tool, many points of consideration necessary to develop successful path planning robots have been encountered. These points were encountered through the need to establish and model the internal modules of work which a robot must do. These sections are necessary in order to reach a stage at which it can make decisions about navigation. Most of these points of consideration, and their effects, have been explained during the design and implementation of the simulator. However, some facets were not obvious until experimental testing was undertaken, when the performance of the simulator and path planning systems could be better judged. During testing the level of flexibility in the settings and desired operation of path planning systems one wishes to simulate, were found to be significant. These factors having the potential to impact on a researcher's progress, through having too many options to choose from and hence encouraging a superfluous number of tests of various combinations.

6.1 Design and Capabilities

As evident by range of trials which were possible to run and the number of parameters which were untouched, this simulation tool provides a large and diverse variety of options for experimental trials and simulations. In addition to parameters and functionality already implemented, due to the modular design the variety

of options can be even further increased through the introduction of other styles of input, control or path planning. Amongst the possible extensions to the current simulator, there is the addition of multiple robots within a terrain, the adaptation to replicate omnidirectional LIDAR input, introducing different forms of noise modelling/distribution and altering the manner of movement.

Modelling sensor types other than LIDAR has not been investigated, but any visual form of sensing should be fairly simple to implement and just draw from the functions available within the Irrlicht library. For other types of sensing and detection it may be necessary to first include modules to produce and model other signals, such as heat or noise and then in turn implement receivers to match such data generators.

Using configuration files in conjunction with shell scripts, it has been possible to run hundreds of unobserved and automated trials with a single command in the terminal. Replication of trials has also been possible, as was necessary in instances when data were lost or an error was encountered. Once the settings to be varied have been selected and altered, the simulation process has been very simple and allowed for multiple trials to be set up within a short period of time and left to progress without prompting or observation.

Though cross-platform usability has not been tested with Microsoft products, porting between different chipsets, structures and operating systems in Linux has been achieved and all of the components used are purported to have cross-platform capabilities. Installation of the simulator, though not yet automated, is fairly simple and without full administration/root privileges it is even possible to install and run a slightly restricted version, which runs minus the SDL output capabilities.

Amongst the capabilities of the simulation tool, is the ability to do basic statistical analysis of data produced as well as recreating/overlaying paths. These two features occur separately from simulation, with each requiring that the simulator is called with a different run option to select the appropriate mode of operation.

6.2 Comparison of Parameters

Monitoring which parameters have been altered is important with a program as diverse and flexible as this one, especially when multiple instances are being run across a number of computers, as within the multitude of choices one can overlook the choices of past values or lose track of how each one interacts with the simulation. An instance of this occurred during testing of the implementation of the LPA* algorithm, when noise inclusion was present following the setup of other trials which were aimed at investigating the effect of noise. The results from the LPA* algorithm when high levels of noise were incorporated, were very poor and initially believed to potentially be a fault or bug with how the LPA* algorithm had been implemented for use on the flexible node network. However, it was then noticed that the inclusion of noise had been left on in the settings. Some form of feedback appears to be necessary to make it easier to discern what variables have been altered and to which values, for a given executable file or running simulation. This is something which would be strongly recommended for future work.

6.3 Flaws and Bugs

At present there are a few bugs in the simulator, the most common of these appears to lie within the Irrlicht graphics engine. Upon creating and setting up a new simulation and graphical window, the simulator exits with an error which indicates that there has been an issue in creating a valid window and as such the GLX driver has terminated the simulation. Running the Valgrind memory leak checker, a couple of warnings are given about system calls with parameters which point to uninitialised addresses, originating within the createWindow function which calls functions from the GLX library. As previously mentioned, there is no feedback from the simulator about internal settings other than those relating to the visual interface, which is an oversight within the design of the simulator that needs to be rectified. The modification of the simulator to take the names and values of parameters it is desired to alter from a new settings input file would in part help

to keep track of which variables were being varied. However, while running a shell script from the terminal window to do multiple trials, it would not possible to easily determine which input files were given. So, while the simulations are still running some form of feedback would still be beneficial, potentially giving the name of the settings file rather than listing all parameters and their values.

6.4 Insight

The usefulness and potential for improving path planning systems has been shown through the experimental trials undertaken. While the results for varying the values of some parameters showed no significant variation, there were cases where a distinct difference in the pattern of results could be observed. One idea which could be taken from an overview of the results gathered, is that for the robot and scenario types modelled, small steps proved to be better than larger steps. Through examining the graphical output at different stages through the traversal of the terrain, observations could also be made about what situations proved the easiest or hardiest in which to identify dangers and also how the network links and node positions affected the followed path.

6.4.1 Config_combos

From the results gathered, the success rates attained using the three Config_combos were distinctly different while the variance between the pair of scenarios within a Config_combos file was minimal in almost all cases, with the exception being the standard version of A* run on the Config_combos_1 setups. Each pair of scenarios had the same positional setup but with different numbers of obstacles inserted, which leads to the observation that for these cases the terrain surface/position appeared to provide the most impact. For the trials run on sim.config(**??**), the reasonably flat surface was easily traversed in a fairly direct path.

Config_combos_2 was the reversal of the start and goal positions used in Config_combos_1. Differences in results between the two configurations are of interest

as the robot is tasked to cover the same ground but in the reverse direction. Looking at where the paths of Config_combos_1 failed in comparison to the successful chosen paths of Config_combos_2, highlights the importance of perspective as descending from a higher position the robot often runs into hazards. Viewing the steep hillside from the bottom, the robot was able to recognise it as a hazard much sooner and hence avoid it by a safe margin, whereas from the top, the robot struggled to gather enough information about the slope until it was very close and at risk of crossing a dangerously steep region.

As the robot builds up its terrain model based on the presence of data, hazards which are best identified by the absence of data (such as steep drops) are not detected as early as they could be. Some form of record of regions the robot has faced but not acquired data from could compliment the model with respect to hazard avoidance. As the altitude of the goal can not be known until it is within visible range and is not occluded by any obstacles, the robot may find itself rising above or descending below the goal due to only being able to base movement on approximate distance in the XY-plane, thus choosing poorly and leading to approaching insurmountable rises or dangerous drops

6.4.2 Movement/Step size

Moving between points, the location of nodes within the network may change to match the alterations within the model due to the gathering of further data about a region as it was approached. The distance travelled between new sets of data being gathered and the frequency with which the planned paths are re-analysed, may affect the speed at which approaching hazards are detected, the quality of path determined and/or the amount of time spent on processing data and planning paths. The setup of the A series of trials examined the effect of varying the maximum number of steps which may be taken between re-analysis (this counter being overridden in the event of nodes changing due to the presence of a hazard). The results across the three sets within the A-series of tests, had very little variation, which may indicate that the number of steps between path reanalysis has little influence or that for the scenarios encountered in the trials,

the detection of new hazards regularly forced early re-analysis for the cases. The recording to file of the count between re-analysis would be a useful addition to determine how often the trials reached the maximum limit for steps before re-analysis.

The trials of A1 and C, further explored the relations of steps through comparing 5 steps of 100 mm to individual steps of 500 mm. While C-trials, for which the maximum number of steps between re-evaluation of paths were low and the steps sizes large, were much faster to run simulations of, they provided poorer results than those from the A series which had smaller steps but more of them potentially occurring between re-evaluation. The large steps resulted in the robot hitting hazards which it had not had a chance to identify yet or overshooting points at which divergence of potential paths may have occurred. In general the large steps did not produce the refined paths exhibited by smaller movement increments, which allow for more curvature within the terrain as opposed to needing long straight regions to traverse. Smaller steps show an increased flexibility in potential paths and greater safety through the increased frequency of hazard checks. This result follows the expectation that restricted/limited movement positions are suboptimal and that a greater flexibility is required. It is felt that this behaviour indicates that flexible networks are better than grid-based networks, which also produce less flexible paths, involving moving regular distances from node to node at the set resolution of the grid.

6.4.3 View/input

The physical qualities of a robotic unit such as its dimensions or the maximum angle it is able to traverse without danger, are obvious factors in affecting potential paths due to defining what is and is not traversable. What can be less obvious is the affect different aspects of the robot's field of view can have. During both the design and testing of the simulation program, the field of vision of the robot was found to have significant affect on the terrain model in a number of ways, and hence the node networks and planned paths as well. The three facets which have a combined affect on the view are the angular range and the maximum distance

within which points were detected, in conjunction with the spread of points within that field. These characteristics of the robot's input view are dependant on the attributes of the sensors, such as the type, location and mounting.

For simplicity and robustness, sensors are often mounted in fixed directions and as such if one is travelling purely in that general direction for the whole trip then the data received should be sufficient to get from start to finish. However, if a deviation is required then very little is known about the regions which it is now necessary to face, although they may have been passed already. Humans have the benefit of interpolating some information about the environment from their peripheral vision and having the capability to look around without needing to stop and change orientation if something interesting is noticed. This difference between man and machine means that the model for an obstacle or hazard encountered in passing may be poorly formed.

For robots with fix mounted sensors the limited angle of view can prove particularly problematic for algorithms such as A* which lack any specific exploratory behaviour as they assume/expect knowledge to have been acquired in all directions. Through having an omnidirectional view, such algorithms can select a drastic change in direction which may lead to moving backwards and away from the goal/direction being faced initially and thus enable passing around an obstacle. When given only a fixed view, the rate at which they may expand their knowledge of peripheral regions is slow and can be hampered by nearby obstacles and hence special cases to re-orient the robot would be highly recommended.

The dispersal of input points received from the sensors is another way in which the sensors can have a strong influence on the quality of the terrain modelling. With the input needing to be discrete rather than continuous, as a unit moves forward along a plane it will tend to get new sets of data which are equally spaced from the previous set. The ratio of the intervals at which the robot moves forwards and the horizontal dispersal of inputs affects the quality/shape of the triangles which constitute the terrain mesh. Even with point culling occurring, the movement's effect on the tessellation can be seen in Figure 6.1 which depicts a sequence of four terrain meshes produced by the robot as it moved. New rows

of points can be seen to be introduced at regular intervals as the robot progresses towards the bottom right corner (sequence A $\rightarrow$ D).

The relative angles between where a robot is situated and the surrounding terrain can also affect the model, as being on an upward or downward slope can limit the information gained if the rest of the terrain is in angled in the opposite direction. This loss of information can be dangerous as it hinders the detection of hazards, which is further discussed in the next section (Section 6.4.4).

Having the robot pan its view across the terrain when it first begins, to get a wider view of its environment was a necessary addition to get the robot to begin to move and consider paths if it is initially faced with a nearby obstacle situated between itself and the destination. For a fixed-vision robot, regularly having it look around itself in directions unrelated to, or perhaps distinctly varied from, the chosen paths may help improve the robot's ability to pick paths through difficult terrain. Similar situations may arise where all the robot's vision is currently filled by a looming obstacle and due to the path of approach the peripheral regions may not have been viewed. The peripheral regions may be occluded by the obstacle being approached, other obstacles or the angle of incline which the robot crossed. Without having viewed a region there can not be network nodes present to draw it into exploring and expanding its field of knowledge and thus regions which may hold potential paths can be missed.

6.4.4 Hazard identification

The hazard identification process becomes more difficult once it is desired that hazardous regions, where applicable, are identified as potentially traversable in certain directions. From identifying potentially traversable hazards, judging at which angles they are safe to approach and cross from, and then representing this information in a useful manner for the path planning system, all processes associated with traversable hazards are laborious. The long term risk of not being able to re-ascend a slope is something which a path planning algorithm must weigh up as it risks becoming locked into a region from which it can not reach its goal, but that is not the only risk involved. While traversing a hazard in a safe direction or

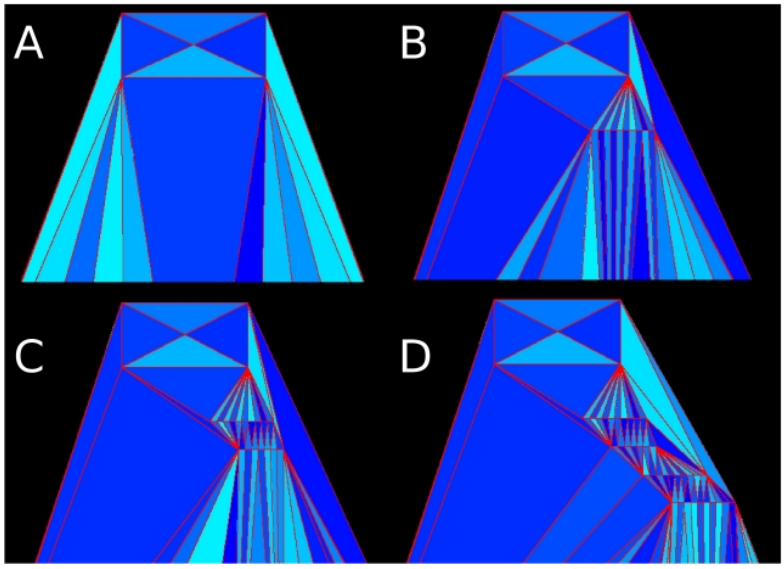

Figure 6.1: Altering Terrain Tessellation as the Robot moves towards bottom right corner (sequence A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D).

position, the path planning system will have to remember the nature of the region being crossed such that it does not attempt to alter direction in order to pursue a newly identified path, as it risks going from being safely orientated to finding itself toppling down a slope or stuck in a crevasse.

Traversable slopes and crevasses both present problems later in the process, when networks of traversable points are formed for use by the path planning algorithms. The basis for forming nodes which link across traversable regions does not apply well when crevasses or directionally traversable slopes are involved and needs some adaptation. The Nodes themselves are also affected in how the unidirectional links are to be represented, as they can be problematic for different algorithms since they use the network of nodes differently. With D* for example, following links in reverse from destination to the current location.

The occurrence of areas which meet hazard identification criteria in all regards except for the size of the faces being too large and hence having low confidence about how accurate they model the terrain in that region, are generally caused through being at the peripheral regions of vision, though sometimes also due to

occlusion by objects in the foreground. Orientating the robot to face the region and gathering data from directly viewing the area of interest provides a means to lift the level of knowledge and would be of benefit for systems which have a greater focus on general exploration, rather than path planning between two positions. The current technique of ignoring the hazard status of large regions works with current goal-driven approach, but does mean an alteration to an exploration-driven system would require significant modification. The removal or increase of the threshold for ignoring the status of a face would be necessary to begin with and then either the addition of functionality within the algorithms to prioritise orientation towards these regions before movement is considered or alternatively the node network sections creating nodes within these sparse areas. The second approach is likely to be more beneficial for pure exploration, whereas the former would work for systems attempting to balance the arrival at a destination, with maximising knowledge acquisition in the process.

Hazardous slopes have been found earlier and more reliably when approaching a steep incline from the base, such as occurred with Config_combos_2, than when on a plateau moving towards a steep drop, like the scenario in Config_combos_1. This is somewhat expected behaviour as a wall is obvious through the presence of data points whereas a drop is shown through an absence and hence may be mistaken for an area which is simply unknown/unscanned.

Curving regions are dangerous as they are modelled with high number of small triangles, which can be difficult to analyse the overall affect they have when they interact/combine as a group. Curves are more of a naturally occurring feature, and it is less likely that they will be found amongst the rubble of man made infrastructure, but one still needs to be aware of their effects. This is potentially why the rectangular obstacles proved less issue than the slopes and rises of the terrain.

6.4.5 Network

The ability to give the robot pre-made/positioned network nodes allows firstly for the testing of the network linking and then secondly the interaction of the path planning algorithm with an idealised network. It is hard for an individual to

discern from a terrain what the correct position for nodes will necessarily be based on the location of hazards and the triangle faces which they are made up of, so this feature allows one to confirm the implementation of an algorithm is working as expected in a theoretical/idealised situation, so that the behaviour upon a system aimed at realism, can be attributed to the more realistic environment and input as opposed to intrinsic to the algorithms implementation.

Further refinement of the node creation and linking, is expected to reduce resources and processing time. However, this must be monitored and a gradual process as it may come at the cost of the quality of the paths produced by the algorithms. Reduction of the distances links span was expected to have minimal impact on the paths, however from comparison of Set H of the experimental results it would appear that the longer links produce better results, although this maybe due to the number and positioning of the nodes. With a modified basis of creation or relative positioning and proximity thresholds, the shorter links may increase to a similar level as networks with links spanning greater distances. Considering the discussion points made earlier in Section 6.4.3, about the effect of the spread of input points received from the sensors, the techniques for creating and linking nodes will also rely on their interaction with the dispersal of input points and the size of the faces created during tessellation. Smaller triangles result in more edges bordering the hazardous and traversable regions, thus increasing the number of hazard nodes and links between them.

In sparse environments, the basis for creating the node network may lead to obstacles being skirted by large distances. The creation of multiple nodes along a line crossing the traversable region between two hazard nodes, may provide an improvement in sparse regions and allow for tighter paths. Creating a fixed number of points upon the lines between hazard nodes does not provide any benefit over a single point as they may lie too close or far apart and as such it would be necessary to make individual calculations based on the distance between specific hazard nodes. Alternatively, allowing a higher number of links and the connection to far off nodes might provide the same behaviour as cutting corners closely. However, it is felt that this would come at the cost of processing and resources.

6.5 Algorithms

In porting the A* and LPA* algorithms for use on a flexible node network, as opposed to the grid-based systems they appear to primarily be used upon, it was found that the re-plan nature of A* was much simpler. For the LPA* algorithm, which follows a re-use approach, the process of updating the node network was difficult due to the potential for nodes to be removed or inserted through improved data for modelling the terrain or identifying hazards. With a grid-based system, updating the network involves polling the status of a node as to whether it is still traversable. However, with a flexible network where the existence of a node signifies that it is traversable it was necessary to instead check for the continued existence of predecessors.

If a predecessor had been removed it was then necessary to determine a new predecessor, however those nodes being considered may have also lost their predecessor and determining which ones to evaluate first is difficult, as they are not set distances apart and hence it is not possible to easily determining what tier or stage they are between the position and goal and hence their natural order. With minimal changes required to port the re-plan approach, one can be more confident about correctness and closeness of the behaviour to the original A* algorithm.

The differences between re-use and re-plan are also likely to have largely been missed due to issues with monitoring the time taken for trials to run, as the main benefit of a re-use approach over a re-plan algorithm is in terms of minimising processing time upon successive iterations. So with this trouble quantifying the level of resource use, it is at present not as beneficial to run trials comparing the different styles of algorithms as it might otherwise be.

Chapter 7

Conclusion

This project has produced a modular simulation tool which allows for the implementation and use of other input types, node networks, or algorithms. In addition to allowing alternative methods of doing core tasks, the simulator provides a variety of options by which the simulation can be altered to identify the affects of different design decisions. The graphical simulation allows users to view and follow the paths chosen by the robot and have a greater understanding of the environment in which it produces these.

For the fixed field of view simulated in the experimental trials, it was found that the role the path planning algorithms played was not the predominant factor but rather one of a set of equally important sections. The nature of the input gathering, the choices made during terrain modelling and the manner in which the node network of potential paths was created, are some other areas which were seen to affect the quality of planned paths.

Initially the intention had only been to allow the testing and comparison of different path planning algorithms, with the parameters being open to change simply due to not knowing the desired values. However, as the project progressed and the influence of other sections became apparent, the scope was expanded to enable the whole system (input/scan, modeling, network creation) to be tested.

The implications of a simulation tool such as this is that the process of design and testing can be made easier and faster, which will make this a valuable tool for

researchers.

The incompatibility of unmodified A* based algorithms with robots that have fixed position sensors with a restricted field of view, was shown. Modification of the behaviour of the algorithms in order to induce the gathering of more information is necessary if fixed view robots are to find paths in difficult terrains. This kind of insight and the ability to test new designs against the current behaviour, are what will make this a useful design tool to aid researchers in designing new systems and algorithms.

Inter-relation between most of the parameters as they combined to form a complete path planning system, makes it hard to identify specific parameters and values as better than others in general, only being able to judge them with respect to the combinations of values of all other parameters under which they were tested. With the number of parameters and the range of values each could potential have, the sheer number of combinations available is immense. As such, the decision of holding certain factors constant and predetermined is required to cut down the number of trials necessary to make design decisions. The level of flexibility provided by this simulation package is both beneficial in the freedom of design it allows but also can have a negative impact on progress as the range of choice in altering parameters entices one to do more comparisons than strictly necessary for one's purposes, and this must be kept in mind when being used.

Overall, the level to which this simulation platform has been developed, with the variety of features and the ability to easily be expanded for use with alternative lines of research, is very high and meets all the ambitions of the initial concept.

7.1 Future work

There is a variety of functionality which is not as at present available, requiring implementation, completion or debugging, which would be suggested as future endeavours.

For the benefit of users, it is highly recommended that some method of ascertaining which parameters had been altered and what the current values are, was

included. This could be done through an option to output to a file the relevant information, or for it to be available and displayed within the GUI.

Functions for generating and using a grid-based node network have been implemented, but testing and improvement are still required as there are bugs present within the code. The completion of this section would enable path planning algorithms such as A*, which were designed to use a limited grid network, to run on their native system.

With regards to the retention and removal of data there are two areas of work which it would be recommended to get to a functional level. Firstly, the use of Hazard objects to reduce the memory requirements of storing distant hazard, is not fully implemented. It is required that the functions for re-inserting old hazards when re-approached, are completed. Having the ability to join and continue hazard objects to previous objects when they cross the threshold/border of retention, is also necessary and should be tested.

The other technique for improving the decisions of keeping or removing data, is the use of confidence levels which are boosted when new data matches the terrain and the application of distance or time based decay upon these confidence levels. With confidence and decay implemented, it would be possible to base the decision to remove data based on how relevant and current it is, by setting a minimum threshold of acceptable confidence values.

One of the most difficult modification which would be recommended, is altering the terrain model from being a single layer of triangle faces, to a model built from isohedrons. Using isohedrons allows for the presence of holes within the terrain model, likely features of a disaster site. Adapting the terrain model is quite simple, primarily requiring a change in the types of CGAL objects used. However, the other modules of the path planning system will most likely require the adaptation of many functions in addition to the creation of some new ones. Having to identify holes, determine whether they are hazardous and then judging where to create mapNode and how to link them will require much thought. The simple methods of displaying information or regions of interest, as two dimensional images will also no longer be sufficient.

A final suggestion would be to create some sort of technique or process by which, the robot can use the absence of data for a region to detect the presence of a dangerous slope or cliff. The inclination of the robot, previous path of approach and the dispersal of scan points would have to be considered in order to not erroneously label a region as a hazard when the lack of data was due to not having viewed a region or due to being occluded. Having a special type of node object for representing regions where there is a lack of information would be a potential way to combine this with the current node network of potential paths.

References

[1] A. Davids, "Urban search and rescue robots: from tragedy to technology," *IEEE Intelligent Systems*, vol. 17, no. 2, pp. 81–83, 2002.

[2] D. Stormont, "Autonomous rescue robot swarms for first responders," in *Computational Intelligence for Homeland Security and Personal Safety, 2005. CIHSPS 2005. Proceedings of the 2005 IEEE International Conference on*, pp. 151–157, 2005.

[3] R. Murphy, "Rescue robotics for homeland security," 2004.

[4] J. Borenstein, G. Granosik, and M. Hansen, "The OmniTread serpentine robot: design and field performance," in *Proceedings of SPIE*, vol. 5804, p. 324, 2005.

[5] D. Hobbelen, T. de Boer, and M. Wisse, "System overview of bipedal robots flame and tulip: Tailor-made for limit cycle walking," in *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems, IROS2008, Nice, France*, pp. 2486–2491, 2008.

[6] LynxMotion, "Beginners guide to pathfinding algorithms." http://www.lynxmotion.com/Category.aspx?CategoryID=100, last visited on 19/12/2009.

[7] C. Ye, S. Ma, and B. Li, "Design and Basic Experiments of a Shape-shifting Mobile Robot for Urban Search and Rescue," in *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 3994–3999, 2006.

[8] R. Murphy, "Marsupial and shape-shifting robots for urban search and rescue," *IEEE Intelligent Systems and Their Applications*, vol. 15, no. 2, pp. 14–19, 2000.

[9] A. Kamimura, H. Kurokawa, E. Yoshida, S. Murata, K. Tomita, and S. Kokaji, "Automatic locomotion design and experiments for a modular robotic system," *IEEE/ASME Transactions on Mechatronics*, vol. 10, no. 3, pp. 314–325, 2005.

[10] D. Williamson and D. Carnegie, "Toward Hierarchical Multi-Robot Urban Search and Rescue: Development of a MotherAgent," *Autonomous robots and agents*, 2007.

[11] H. Ishida, K. Nagatani, and Y. Tanaka, "Three-dimensional localization and mapping for a crawler-type mobile robot in an occluded area using the scan matching method," in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings*, vol. 1.

[12] J. Carlson, R. Murphy, and A. Nelson, "Follow-up analysis of mobile robot failures," in *2004 IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA'04*, vol. 5.

[13] R. Bostelman, T. Hong, R. Madhavan, and B. Weiss, "3D range imaging for urban search and rescue robotics research," in *2005 IEEE International Safety, Security and Rescue Robotics, Workshop*, pp. 164–169, 2005.

[14] A. Jacoff, E. Messina, B. Weiss, S. Tadokoro, and Y. Nakagawa, "Test arenas and performance metrics for urban search and rescue robots," in *2003 IEEE/RSJ International Conference on Intelligent Robots and Systems, 2003.(IROS 2003). Proceedings*, vol. 4, 2003.

[15] Defense Advanced Research Projects Agency, "Darpa grand challenge." http://www.darpa.mil/grandchallenge/index.asp.

[16] S. Wirth and J. Pellenz, "Exploration Transform: A stable exploring algorithm for robots in rescue environments," in *Workshop on Safety, Security, and Rescue Robotics*, 2007.

[17] J. Craighead, R. Murphy, J. Burke, and B. Goldiez, "A Survey of Commercial & Open Source Unmanned Vehicle Simulators," in *2007 IEEE International Conference on Robotics and Automation*, pp. 852–857, 2007.

[18] B. Balaguer, S. Balakirsky, S. Carpin, M. Lewis, and C. Scrapper, "USARSim: a validated simulator for research in robotics and automation," in *Workshop on Robot Simulators: Available Software, Scientific Applications, and Future Trends at IEEE/RSJ*, 2008.

[19] B. Delaunay, "Sur la sphere vide," *Izv. Akad. Nauk SSSR, Otdelenie Matematicheskii i Estestvennyka Nauk*, vol. 7, pp. 793–800, 1934.

[20] S. Rebay, "Efficient unstructured mesh generation by means of Delaunay triangulation and Bowyer-Watson algorithm," *Journal of Computational Physics*, vol. 106, no. 1, pp. 125–138, 1993.

[21] J. Lee and R. Dyczij-Edlinger, "Automatic mesh generation using a modified Delaunay tessellation," *IEEE Antennas and Propagation Magazine*, vol. 39, no. 1, pp. 34–45, 1997.

[22] T. Gerbaud, V. Polotski, and P. Cohen, "Simultaneous exploration and 3D mapping of unstructured environments," in *2004 IEEE International Conference on Systems, Man and Cybernetics*, vol. 6, 2004.

[23] H. Durrant-Whyte and T. Bailey, "Simultaneous Localisation and Mapping (SLAM): Part I The Essential Algorithms," *Robotics and Automation Magazine*, vol. 13, no. 2, pp. 99–110, 2006.

[24] J. M. Holland, *Designing mobile autonomous robots*. Elsevier Newnes, 2004.

[25] A. T. de.Almeida and O. Khatib, *Autonomous robotic systems: - Advanced Research Workshop on Autonomous Robotic Systems*. Springer, 1998.

[26] M. K. Miller, N. Winkless, and J. Bosworth, *The personal robot navigator*. Robot Press, 1998.

[27] S. Iyengar and A. Elfes, *Autonomous mobile robots*. IEEE Computer Society Press, 1991.

[28] R. M. Jensen, R. E. Bryant, and M. M. Veloso, "SetA*: An efficient BDD-based heuristic search algorithm," tech. rep., Computer Science Department, Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh,PA, August 2002.

[29] "Beginners guide to pathfinding algorithms." http://ai-depot.com/Tutorial/PathFinding.html, visited on 3/9/2006.

[30] J. Pearl, *Heuristics–intelligent search strategies for computer problem solving*. Addison-Wesley Publishing Co., Reading, MA, 1984.

[31] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to algorithms*. The MIT press, 2001.

[32] P. Hart, N. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[33] K. Manley, "Pathfinding: from A* to LPA," *University of Minnesota*, 2003.

[34] A. Stentz, "Optimal and efficient path planning for partially-known environments," in *Robotics and Automation, 1994. Proceedings., 1994 IEEE International Conference on*, pp. 3310–3317 vol.4, May 1994.

[35] S. Koenig and M. Likhachev, "D* Lite," in *Proceedings of the National Conference on Artificial Intelligence*, pp. 476–483, Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2002.

[36] D. Ferguson and A. Stentz, "Multi-resolution Field D*," in *Proceedings of the International Conference on Intelligent Autonomous Systems (IAS)*, 2006.

[37] J. Wang, M. Lewis, and J. Gennari, "Interactive simulation of the NIST USAR arenas," in *IEEE International Conference on Systems, Man and Cybernetics, 2003*, vol. 2, 2003.

[38] M. Kadous, R. Sheh, and C. Sammut, "Controlling Heterogeneous Semi-autonomous Rescue Robot Teams," in *IEEE International Conference on Systems, Man and Cybernetics, 2006. SMC'06*, vol. 4, 2006.

[39] Nikolaus Gebhardt & Development Team, "Irrlicht 3d graphics engine." http://irrlicht.sourceforge.net/.

[40] D. Garlan and M. Shaw, "An introduction to software architecture," Tech. Rep. CMU-CS-94-166, Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh,PA, January 1994.

[41] B. Chazelle, "Triangulating a simple polygon in linear time," *Discrete and Computational Geometry*, vol. 6, no. 1, pp. 485–524, 1991.

[42] R. Gilberg and B. Forouzan, *Data structures: a pseudocode approach with C++*. Brooks/Cole Publishing Co. Pacific Grove, CA, USA, 2001.

[43] CGAL Development Team, "Computational geometry algorithms library." http://www.cgal.org/.

[44] J. hoey, "Wheelchair obstacle avoidance helper (woah) system." http://www.cs.toronto.edu/ jhoey/wheel/mapping.html.

Appendix A

Source Code for Simulation System

A.1 Algorithms

A.2 Algorithms.h

```
1  /*************************************************************************
2   *   Copyright (C) 2009 by Michael Douglas Hasler   *
3   *   redheadpunk@gmail.com   *
4   *                                                                       *
5   *   This program is free software; you can redistribute it and/or modify  *
6   *   it under the terms of the GNU General Public License as published by  *
7   *   the Free Software Foundation; either version 2 of the License, or     *
8   *   (at your option) any later version.                                   *
9   *                                                                       *
10  *   This program is distributed in the hope that it will be useful,       *
11  *   but WITHOUT ANY WARRANTY; without even the implied warranty of        *
12  *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the         *
13  *   GNU General Public License for more details.                          *
14  *                                                                       *
15  *   You should have received a copy of the GNU General Public License     *
16  *   along with this program; if not, write to the                         *
17  *   Free Software Foundation, Inc.,                                       *
18  *   59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.             *
19  *************************************************************************/
20 #ifndef ALG_H_
21 #define ALG_H_
22
23 class Robot;
24 #include "World.h"
```

```
25 #include "Robot.h"
26
27 enum STATE_NAME{
28     FORWARD = 0,
29     LEFT,
30     RIGHT,
31     BACK,
32     FWDLEFT,
33     FWDRIGHT
34 };
35
36 const float H_WEIGHTING = 1.0;
37 const float SELECTION_RANGE = 5000.0f;//should usually be set to same as
       visi_dist of robot
38 STATE_NAME currentState = FORWARD;
39 STATE_NAME fwdSide = FWDLEFT;
40
41     /**
42         A_star needs open/closed bool, f,g & h scores and parent link
43
44         get Node, calculate f[] then add to openSet
45         pick lowest f[] in openset
46         look at the nodes linked to by pick
47             if in closedSet ignore,
48             if not in open add to open (with pick as parent) do f[],g[],h[]
49             if in open set, compare based on g[], if lower g[] then set pick as
                   parent and recalc f[],g[],h[]
50
51         exit when goal is added to closedSet or openSet is empty
52     */
53     template <class NODE>
54     void A_Star_Ctrl(Robot* robot)
55     {
56         Point next = robot->GetNext();
57         Coord shift = robot->MoveDirection();
58         if( !(shift.x() == 0 && shift.y() == 0 && shift.z() == 0) && robot->
               numMoves < STEPS_BETWEEN_EVAL && robot->terrain->GetCurrentNode()->
               GetPoint() != next)
59             robot->numMoves++;
60         else
61         {
62             NODE* currentLocation, *goalLocation;
63             robot->terrain->AllocateNodeArrays();
64             robot->GetMapNodes<NODE>(&currentLocation, &goalLocation);//curr loc
                   return with parent?
65             robot->numMoves = 0;
66             int numPathNodes = 1;
```

```
67              NODE* tempNode = A_Star_Eval(robot, currentLocation, goalLocation);
68              NODE* tNode = tempNode;
69              while(tNode != currentLocation) //tNode->Parent != NULL
70              {
71                  tNode = tNode->predecessor;
72                  numPathNodes++;
73              }
74
75              robot->SetNumSteps(numPathNodes);
76              Point* path = (Point*) calloc(numPathNodes, sizeof(Point));//this
                    affects the currentPos how/why?
77
78              for( int l = numPathNodes - 1; l >= 0; l--)
79              {
80                  path[l] = tempNode->GetPoint();
81                  tempNode = tempNode->predecessor;
82              }
83              robot->SetPath(path);
84              free(path);
85              path = NULL;
86          }
87      }
88
89      /** Have each algorithm having its own node type which extends mapNode?
90
91          A_star needs open/closed bool, f,g & h scores and parent link
92
93          //get Node, calculate f[] then add to openSet
94          //pick lowest f[] in openset
95          //look at the nodes linked to by pick
96          //  if in closedSet ignore,
97          //  if not in open add to open (with pick as parent) do f[],g[],h[]
98          //  if in open set, compare based on g[], if lower g[] then set pick as
                parent and recalc f[],g[],h[]
99
100         //exit when goal is added to closedSet or openSet is empty
101
102     */
103     template <class NODE>
104     NODE* A_Star_Eval(Robot* robot, NODE* currentLocation, NODE* goalLocation)
105     {
106         int numNodes = 0;
107         int openI = 0, closedI = 0;
108         NODE** closedSet = (NODE**) calloc(ARRAY_CHUNK, sizeof(NODE*));
109         NODE** openSet = (NODE**) calloc(ARRAY_CHUNK, sizeof(NODE*));
110         NODE* goal = goalLocation;
111         NODE* start = currentLocation;
```

```
112         NODE *currentPick, *closest;
113
114         start->g_val = 0;
115         start->h_val = CalculateDirectDistance(start->GetPoint(), goal->GetPoint
                ), false);
116         start->f_val = start->h_val;
117         start->closed = true;
118         start->open = false;
119         currentPick = start;
120         closest = start;
121
122         do
123         {
124             for(int i = 0; i < currentPick->GetNumLinks(); i++)
125             {
126                 NODE* temp = currentPick->FollowLink(i);
127                 if(temp->closed)
128                     continue;
129                 else if(!temp->open)
130                 {
131                     temp->open = true;
132                     temp->closed = false;
133                     temp->predecessor = currentPick;
134                     if(openI%ARRAY_CHUNK == 0 && openI != 0)
135                     {
136                         void* tmp = realloc(openSet, (openI + ARRAY_CHUNK) *
                                sizeof(NODE*));
137                         if(tmp != NULL)
138                             openSet = (NODE**) tmp;
139                     }
140
141                     openSet[openI++] = temp;
142                     temp->g_val = currentPick->g_val + CalculateDirectDistance(
                            currentPick->GetPoint(), temp->GetPoint(), false);
143                     temp->h_val = CalculateDirectDistance(temp->GetPoint(), goal
                            ->GetPoint(),false) * H_WEIGHTING;/* weighting is done
                            against h_val*/
144                     temp->f_val = temp->g_val + temp->h_val;
145                 }
146                 else
147                 {
148                     float alt_g = currentPick->g_val + CalculateDirectDistance(
                            currentPick->GetPoint(), temp->GetPoint(), false);
149                     if(alt_g < temp->g_val)
150                     {
151                         temp->predecessor = currentPick;
152                         temp->g_val = alt_g;
```

```
153                     temp->f_val = temp->g_val + temp->h_val;
154                 }
155             }
156         }
157
158         //quicksort .. of sorts
159         int increment = 3;//sorts in buckets/clumps of three
160         NODE* tempNode;
161         while(increment > 0)
162         {
163             for(int j = 0; j < openI; j++)
164             {
165                 int k = j;
166                 tempNode = openSet[j];
167                 while ((k >= increment) && (openSet[k-increment]->f_val <
                        tempNode->f_val))
168                 {
169                     openSet[k] = openSet[k - increment];
170                     k = k - increment;
171                 }
172                 openSet[k] = tempNode;
173             }
174             if(increment/2 != 0)
175                 increment = increment/2;
176             else if(increment == 1)
177                 increment = 0;
178             else
179                 increment = 1;
180         }
181
182         if(openI > 0)
183         {
184             currentPick = openSet[--openI];
185             currentPick->open = false;
186             currentPick->closed = true;
187             openSet[openI] = NULL;
188
189             if(currentPick->h_val < closest->h_val || closest == start)
190                 closest = currentPick;
191
192             if(closedI%ARRAY_CHUNK == 0 && closedI != 0)
193             {
194                 void* tmp = realloc(closedSet, (closedI + ARRAY_CHUNK) *
                        sizeof(NODE*));
195                 if(tmp != NULL)
196                     closedSet = (NODE**) tmp;
197             }
```

```
198                 closedSet[closedI++] = currentPick;
199             }
200         }
201         while(openI > 0 && !goal->closed);
202 
203         int numPathNodes = 1;
204         NODE* tempNode;
205         Point* path;
206 
207         if(goal->closed)
208             tempNode = goal;
209         else
210         {
211             if(SHOW_WARNINGS) printf("Warning - The open set is empty yet Goal
                    has not been added to the closed set: No full path found\n");
212             else      printf("Incomplete path\n");
213 
214             tempNode = closest;//could lead to being stuck in dead ends.
215         }
216 
217         for(int i = 0; i < openI; i++)
218             openSet[i]->ResetOpenClosed();
219 
220         for(int i = 0; i < closedI; i++)
221             closedSet[i]->ResetOpenClosed();
222 
223         free(closedSet);
224         free(openSet);
225         closedSet = NULL;
226         openSet = NULL;
227 
228         return tempNode;
229     }
230 
231     /** LPA_star needs open/closed bool, f,g & h scores and parent link
232         Move STEPS_BETWEEN_EVAL or to next node, before re-generating nodes
233         Upon regen, only recalc path if at next node or current path is no longer
                valid
234     */
235     template <class NODE>
236     void LPA_Star(Robot* robot)
237     {
238         Point next = robot->GetNext();
239         if(robot->numMoves < STEPS_BETWEEN_EVAL && robot->terrain->GetCurrentNode
                ()->GetPoint() != next)
240             robot->numMoves++;
241         else
```

```
242         {
243             bool reAnalyse = true;
244             robot->numMoves = 0;
245             NODE* currentLocation, *goalLocation;
246             robot->terrain->AllocateNodeArrays();
247             robot->GetMapNodes<NODE>(&currentLocation, &goalLocation);
248             if(currentLocation->GetPoint() != next)
249             {
250                 for(int i = 0; i < currentLocation->GetNumLinks(); i++)//numLinks
                        always 0, hasn't been linked yet?
251                 {
252                     if(currentLocation->FollowLink(i)->GetPoint() == next)
253                     {
254                         NODE* temp = robot->terrain->GetGoalNode();
255                         if(temp->g_val == -1)
256                             reAnalyse = false;//if goal is not yet linked, then
                                skip updating graph -- but if don't update, how
                                will it become linked
257                         //check predecessors
258                         //if fine then continue, otherwise want graph updated
259                         while(reAnalyse && temp->predecessor != NULL && !temp->
                            predecessor->CheckForMark() )
260                         {
261                             temp = temp->predecessor;
262                             if(temp->GetPoint() == next)
263                                 reAnalyse = false;//if path to goal is fine, then
                                    skip updating graph
264                         }
265                         break;
266                     }
267                 }
268             }
269
270             if(reAnalyse)
271                 UpdateLPA(robot, currentLocation, goalLocation);
272         }
273     }
274
275     /**
276         //get Node, calculate f[] then add to openSet
277         //pick lowest f[] in openset
278         //look at the nodes linked to by pick
279         //  if in closedSet ignore,
280         //  if not in open add to open (with pick as parent) do f[],g[],h[]
281         //  if in open set, compare based on g[], if lower g[] then set pick as
                parent and recalc f[],g[],h[]
282
```

```
283         //exit when goal is added to closedSet or openSet is empty
284
285     */
286     template <class NODE>
287     void UpdateLPA(Robot* robot, NODE* currentLocation, NODE* goalLocation)
288     {
289         int numNodes = 0;
290         int* openI = &robot->terrain->openInd;
291         int* closedI = &robot->terrain->closedInd;
292         NODE*** closedSet = &robot->terrain->closedSet;
293         NODE*** openSet = &robot->terrain->openSet;
294         NODE *currentPick, *closest;
295
296         currentLocation->g_val = 0;
297         currentLocation->h_val = CalculateDirectDistance(currentLocation->
                GetPoint(), goalLocation->GetPoint(), false);
298         currentLocation->f_val = currentLocation->h_val;
299         currentLocation->closed = true;
300         currentLocation->open = false;
301         closest = currentLocation;
302         currentPick = currentLocation;
303
304 //      UpdateSets(openSet, closedSet, &openI, &closedI);//Alternate approach to
        ported behaviour - less tested
305         UpdateSetsAndClear(openSet, closedSet, openI, closedI);
306
307         do
308         {
309             for(int i = 0; i < currentPick->GetNumLinks(); i++)
310             {
311                 NODE* temp = currentPick->FollowLink(i);
312
313                 if(temp->closed)
314                     continue;
315                 else
316                 {
317                     if(!temp->open)
318                     {
319                         temp->open = true;
320                         temp->closed = false;
321
322                         if((*openI)%ARRAY_CHUNK == 0 && *openI != 0)
323                         {
324                             void* tmp = realloc(*openSet, (*openI + ARRAY_CHUNK)
                                    * sizeof(NODE*));
325                             if(tmp != NULL)
326                                 *openSet = (NODE**) tmp;
```

```
327                         }
328
329                         (*openSet)[(*openI)++] = temp;
330                     }
331
332                     if(temp->g_val == -1)
333                     {
334                         temp->predecessor = currentPick;
335                         temp->g_val = currentPick->g_val +
                                CalculateDirectDistance(currentPick->GetPoint(), temp
                                ->GetPoint(), false);
336                         temp->h_val = CalculateDirectDistance(temp->GetPoint(),
                                goalLocation->GetPoint(),false) * H_WEIGHTING;/*
                                weighting is done against h_val*/
337                         temp->f_val = temp->g_val + temp->h_val;
338                     }
339                     else
340                     {
341                         float alt_g = currentPick->g_val +
                                CalculateDirectDistance(currentPick->GetPoint(), temp
                                ->GetPoint(), false);
342                         if(alt_g < temp->g_val || temp->predecessor == NULL)
343                         {
344                             float adj = alt_g - temp->g_val;
345                             temp->predecessor = currentPick;
346                             AdjustSuccessors(temp, adj);
347                         }
348
349                     }
350                 }
351             }
352
353             int increment = 3;//sorts in buckets/clumps of three
354             int switchTop = *openI;
355             NODE* tempNode;
356             while(increment > 0)
357             {
358                 for(int j = 0; j < *openI; j++)
359                 {
360                     int k = j;
361                     tempNode = (*openSet)[j];
362                     while((tempNode == NULL || !tempNode->open) && *openI > j)
363                     {
364                         (*openSet)[j] = (*openSet)[--(*openI)];
365                         tempNode = (*openSet)[j];
366                     }
367
```

```
368                     while (k >= increment && (*openSet)[k-increment]->f_val <
                            tempNode->f_val)
369                     {
370                         (*openSet)[k] = (*openSet)[k - increment];
371                         k = k - increment;
372                     }
373                     (*openSet)[k] = tempNode;
374                 }
375                 if(increment/2 != 0)
376                     increment = increment/2;
377                 else if(increment == 1)
378                     increment = 0;
379                 else
380                     increment = 1;
381             }
382
383             if(*openI > 0)
384             {
385                 currentPick = (*openSet)[--(*openI)];
386                 currentPick->open = false;
387                 currentPick->closed = true;
388
389                 while(!currentPick->predecessor->closed && *openI > 0)
390                 {
391                     if(!UpdatePredecessors(currentPick, closedSet, closedI))
392                     {
393                         currentPick = (*openSet)[--(*openI)];
394                         currentPick->open = false;
395                         currentPick->closed = true;
396                     }
397                 }
398
399                 if(*openI >= 0)
400                 {
401                     (*openSet)[*openI] = NULL;
402                     if(closest == currentLocation || currentPick->h_val < closest
                            ->h_val)
403                         closest = currentPick;
404
405                     if((*closedI)%ARRAY_CHUNK == 0 && *closedI != 0)
406                     {
407                         void* tmp = realloc(*closedSet, (*closedI + ARRAY_CHUNK)
                                * sizeof(NODE*));
408                         if(tmp != NULL)
409                             *closedSet = (NODE**) tmp;
410                     }
411                     (*closedSet)[(*closedI)++] = currentPick;
```

```
412
413                     int j = 0;
414                     for(j = 0; j < *closedI && currentPick->predecessor !=
                            currentLocation; j++)
415                     {
416                         if((*closedSet)[j] == currentPick->predecessor || (*
                                closedSet)[j] == NULL)
417                         {
418                             break;
419                         }
420                     }
421                 }
422             }
423         }
424         while(*openI > 0 && !goalLocation->closed);
425
426         int numPathNodes = 1;
427         NODE* tempNode;
428         Point* path;
429
430         if(goalLocation->closed)
431             tempNode = goalLocation;
432         else
433         {
434             if(SHOW_WARNINGS) printf("Warning - The open set is empty yet Goal
                    has not been added to the closed set: No full path found\n");
435             else      printf("Incomplete path\n");
436
437             tempNode = closest;//could lead to being stuck in dead ends. needs
                    exploration mode or something when no path
438         }
439
440         NODE* tNode = tempNode;
441         while(tNode != currentLocation)
442         {
443             if(tNode->predecessor == NULL)
444             {
445                 if(tempNode != closest)
446                 {
447                     tNode = closest;
448                     tempNode = closest;
449                     numPathNodes = 0;
450                 }
451                 else
452                 {
453                     if(robot->numBadPath++ == 5)
454                     {
```

```
455                         UpdateSetsAndClear(openSet, closedSet, openI, closedI);
456                         robot->numBadPath = 0;
457                     }
458 
459                     return;
460                 }
461             }
462             else
463             {
464                 tNode = tNode->predecessor;
465                 numPathNodes++;
466             }
467         }
468 
469         robot->SetNumSteps(numPathNodes);
470         path = (Point*) calloc(numPathNodes, sizeof(Point));
471         for( int l = numPathNodes - 1; l >= 0; l--)
472         {
473             path[l] = tempNode->GetPoint();
474             tempNode = tempNode->predecessor;
475         }
476         robot->SetPath(path);
477         free(path);
478         path = NULL;
479     }
480 
481     /**
482         //check nodes in open and closed sets
483         //if open set now has prec = null, remove from open set
484         //if open set now has prec != null, no change
485         //if closed set now has prec = null, remove from closed set
486             //if succ != null, recursively remove them from closed or open sets
487         //if closed set now has succ = null, move to open set
488 
489         //compact sets and sort/order the open one
490     */
491     template <class NODE>
492     void UpdateSets(NODE*** openSet, NODE*** closedSet, int* openI, int* closedI)
493     {
494         int openInsertInd = 0;
495         int closedInsertInd = 0;
496         for(int i = 0; i < *closedI; i++)
497         {
498             if((*closedSet)[i] == NULL || (*closedSet)[i]->g_val == 0)
499                 continue;
500             else if((*closedSet)[i]->predecessor == NULL)
501             {
```

```
502                 UpdateNode((*closedSet)[i]);
503             }
504             else if((*closedSet)[i]->CheckForMark() || !(*closedSet)[i]->closed)
505             {
506 //                  if((*closedSet)[i]->closed)
507 //                  {
508 //                      (*closedSet)[i]->closed = false;
509 //                          if((*closedSet)[i]->successor != NULL)
510 //                          UpdateSuccessors((*closedSet)[i]);
511 //                  }
512                 (*closedSet)[i] = NULL;
513             }
514             else if((*closedSet)[i]->successor == NULL)
515             {
516                 bool newChild = false;
517                 for(int j = 0; j < (*closedSet)[i]->GetNumLinks(); j++)
518                 {
519                     NODE* temp = (*closedSet)[i]->FollowLink(j);
520                     if(!temp->closed && !temp->open && (temp->predecessor == NULL
                            || temp->predecessor == (*closedSet)[i]))
521                         newChild = true;
522                 }
523
524                 if(newChild)
525                 {
526                     if((*openI)%ARRAY_CHUNK == 0 && *openI != 0)
527                     {
528                         void* tmp = realloc(*openSet, (*openI + ARRAY_CHUNK) *
                                sizeof(NODE*));
529                         if(tmp != NULL)
530                             *openSet = (NODE**) tmp;
531                     }
532
533                     (*closedSet)[i]->open = true;
534                     (*closedSet)[i]->closed = false;
535                     (*openSet)[(*openI)++] = (*closedSet)[i];
536                     (*closedSet)[i] = NULL;
537                 }
538             }
539         }
540
541         for(int i = 0; i < *closedI; i++)
542         {
543             if((*closedSet)[i] != NULL && (*closedSet)[i]->closed)
544                 (*closedSet)[closedInsertInd++] = (*closedSet)[i];
545         }
546         *closedI = closedInsertInd;
```

```
547
548         for(int i = 0; i < *openI; i++)
549         {
550             if((*openSet)[i] != NULL && (*openSet)[i]->predecessor == NULL)
551                 UpdateNode((*openSet)[i]);
552         }
553
554         for(int i = 0; i < *openI; i++)
555         {
556             if((*openSet)[i] == NULL)
557                 continue;
558             else if((*openSet)[i]->CheckForMark() || !(*openSet)[i]->open || (*
                    openSet)[i]->predecessor == NULL)
559             {
560                 (*openSet)[i]->open = false;
561                 (*openSet)[i] = NULL;
562             }
563             else
564                 (*openSet)[openInsertInd++] = (*openSet)[i];
565         }
566         *openI = openInsertInd;
567
568         if(openInsertInd < *openI)
569         {
570             int increment = 3;//sorts in buckets/clumps of three
571             NODE* tempNode;
572             while(increment > 0)
573             {
574                 for(int j = 0; j < *openI; j++)
575                 {
576                     int k = j;
577                     tempNode = (*openSet)[j];
578                     while ((k >= increment) && ((*openSet)[k-increment]->f_val <
                            tempNode->f_val))
579                     {
580                         (*openSet)[k] = (*openSet)[k - increment];
581                         k = k - increment;
582                     }
583                     (*openSet)[k] = tempNode;
584                 }
585                 if(increment/2 != 0)
586                     increment = increment/2;
587                 else if(increment == 1)
588                     increment = 0;
589                 else
590                     increment = 1;
591             }
```

```
592         }
593     }
594
595     /**
596         //check nodes in open and closed sets, updating if necessary
597         //if open set now has prec = null, Update
598         //if closed set now has prec = null, update
599         // Remove all nodes from the sets
600     */
601     template <class NODE>
602     void UpdateSetsAndClear(NODE*** openSet, NODE*** closedSet, int* openI, int*
            closedI)
603     {
604         ///if(parent = NULL, then remove from open || closed
605         bool closedNullHit = false;
606         bool openNullHit = false;
607         bool sepClosedFirst = false;
608         bool sepOpenFirst = false;
609         int firstClosedNull = -1;
610         int firstOpenNull = -1;
611         int lastClosedNull = -1;
612         int lastOpenNull = -1;
613         for(int i = 0; i < *closedI; i++)
614         {
615             if((*closedSet)[i] == NULL || (*closedSet)[i]->g_val == 0)
616                 continue;
617             else if((*closedSet)[i]->predecessor == NULL)
618             {
619                 UpdateNode((*closedSet)[i]);
620             }
621         }
622
623         for(int i = 0; i < *closedI; i++)
624         {
625             if((*closedSet)[i] != NULL)
626             {
627                 (*closedSet)[i]->closed = false;
628                 (*closedSet)[i] = NULL;
629             }
630         }
631
632         for(int i = 0; i < *openI; i++)
633         {
634             if((*openSet)[i] != NULL && (*openSet)[i]->predecessor == NULL)
635                 UpdateNode((*openSet)[i]);
636         }
637
```

```
638         for(int i = 0; i < *openI; i++)
639         {
640             if((*openSet)[i] != NULL)
641             {
642                 (*openSet)[i]->open = false;
643                 (*openSet)[i] = NULL;
644             }
645         }
646 
647         *openI = 0;
648         *closedI = 0;
649 
650         void* tmp = realloc(*closedSet, ARRAY_CHUNK * sizeof(NODE*));
651         if(tmp != NULL)
652             *closedSet = (NODE**) tmp;
653         tmp = realloc(*openSet, ARRAY_CHUNK * sizeof(NODE*));
654         if(tmp != NULL)
655             *openSet = (NODE**) tmp;
656     }
657 
658     /** Compare a node to adjacent ones to see whether it is locally consistent.
659         Find lowest g_val, if is equal or lower, then make sure predecessor
                points to that link and adjust values as necessary
660         If not, for higher rhs recursively go through til nodes are made locally
                consistent
661         After updating check to see if has viable predecessor
662     */
663     template <class NODE>
664     void UpdateNode(NODE* testNode)
665     {
666         float originalG;
667         float lowestG = -1;
668         int linkNumLowG = -1;
669 
670         FindLowestG(testNode, &lowestG, &linkNumLowG);
671         if(lowestG > -1)
672         {
673             float rhs = CalculateDirectDistance(testNode->GetPoint(), testNode->
                    FollowLink(linkNumLowG)->GetPoint(), true) + lowestG;
674             if(testNode->g_val < rhs)
675             {
676                 originalG = testNode->g_val;
677                 testNode->g_val = -1;
678                 for(int j = 0; j < testNode->GetNumLinks(); j++)
679                 {
680                     if(testNode->FollowLink(j)->predecessor == testNode)
681                         UpdateNode(testNode->FollowLink(j));
```

```
682                 }
683
684                 lowestG = -1;
685                 linkNumLowG = -1;
686                 FindLowestG(testNode, &lowestG, &linkNumLowG);
687
688                 if(lowestG > -1)
689                 {
690                     rhs = CalculateDirectDistance(testNode->GetPoint(), testNode
                        ->FollowLink(linkNumLowG)->GetPoint(), true) + lowestG;
691                     if(testNode->FollowLink(linkNumLowG)->predecessor == testNode
                        )
692                     {
693 //                      testNode->FollowLink(linkNumLowG)->predecessor = NULL;
694                         UpdateNode(testNode->FollowLink(linkNumLowG));
695                     }
696
697                     testNode->predecessor = testNode->FollowLink(linkNumLowG);
698                     if(originalG != rhs)
699                     {
700                         testNode->g_val = originalG;
701                         AdjustSuccessors(testNode, rhs - originalG);
702                     }
703                 }
704                 else
705                 {
706                     testNode->predecessor = NULL;
707                     testNode->open = false;
708                     testNode->closed = false;
709                 }
710             }
711             else
712             {
713                 testNode->predecessor = testNode->FollowLink(linkNumLowG);
714                 if(testNode->g_val > rhs)
715                     AdjustSuccessors(testNode, rhs - testNode->g_val);
716             }
717         }
718         else
719         {
720             testNode->g_val = -1;
721             testNode->open = false;
722             testNode->closed = false;
723             testNode->predecessor = NULL;
724             testNode->successor = NULL;
725         }
726     }
```

```
727
728     /** Find the linked neighbour with the lowest g-value */
729     template <class NODE>
730     void FindLowestG(NODE* testNode, float* lowestG, int* linkNumLowG)
731     {
732         /** Check non-successors first and then only if no other choice try
                successors **/
733         bool repeat = false;
734         do
735         {
736             for(int i = 0; i < testNode->GetNumLinks(); i++)
737             {
738                 if(testNode->FollowLink(i)->g_val == -1)
739                     continue;
740                 else if(testNode->FollowLink(i)->predecessor != testNode && (*
                        lowestG == -1 ||  testNode->FollowLink(i)->g_val < *lowestG
                        || (testNode->FollowLink(i)->g_val == *lowestG && testNode->
                        FollowLink(i)->h_val < testNode->FollowLink(*linkNumLowG)->
                        h_val) ))
741                 {
742                     *lowestG = testNode->FollowLink(i)->g_val;
743                     *linkNumLowG = i;
744                 }
745             }
746
747             if(!repeat && *lowestG == -1)
748             {
749                 repeat = true;
750                 testNode->g_val = -1;
751                 for(int j = 0; j < testNode->GetNumLinks(); j++)
752                 {
753                     if(testNode->FollowLink(j)->predecessor == testNode)
754                         UpdateNode(testNode->FollowLink(j));
755                 }
756             }
757             else
758                 repeat = false;
759
760             if(*linkNumLowG > -1 && testNode->FollowLink(*linkNumLowG)->
                    predecessor == testNode)
761             {
762                 testNode->predecessor = NULL;
763                 testNode->FollowLink(*linkNumLowG)->predecessor = NULL;
764             }
765         }
766         while(repeat);
767     }
```

```
768
769     /** Remove successive nodes from the open and closed sets */
770     template <class NODE>
771     void UpdateSuccessors(NODE* temp)
772     {
773         temp->successor->closed = false;
774         temp->successor->open = false;
775         if(temp->successor->successor != NULL)
776             UpdateSuccessors(temp->successor);
777     }
778
779     /** Propagate changes to predecessors */
780     template <class NODE>
781     bool UpdatePredecessors(NODE* temp,  NODE*** closedSet, int* closedI)
782     {
783         bool result = true;
784         if(!temp->predecessor->closed)
785         {
786             temp->predecessor->closed = true;
787             temp->predecessor->open = false;
788             if((*closedI)%ARRAY_CHUNK == 0 && *closedI != 0)
789             {
790                 void* tmp = realloc(*closedSet, (*closedI + ARRAY_CHUNK) * sizeof
                        (NODE*));
791                 if(tmp != NULL)
792                     *closedSet = (NODE**) tmp;
793             }
794             (*closedSet)[(*closedI)++] = temp->predecessor;
795
796             if(temp->predecessor->predecessor != NULL)
797                 result = UpdatePredecessors(temp->predecessor, closedSet, closedI
                        );
798             else if(temp->predecessor->g_val != 0)
799                 result = false;
800         }
801
802         return result;
803     }
804
805     /** Update values of successive nodes */
806     template <class NODE>
807     void AdjustSuccessors(NODE* temp, float adjustment)
808     {
809         if(temp->g_val < 0 && adjustment < 0)
810             int cat = 7;
811         temp->g_val += adjustment;
812         temp->f_val = temp->g_val + temp->h_val;
```

```
813         if(temp->g_val < 0)
814             int cat = 7;
815 
816         for(int i = 0; i < temp->GetNumLinks(); i++)
817         {
818             if(temp->FollowLink(i)->predecessor == temp)
819             {
820                 AdjustSuccessors(temp->FollowLink(i), adjustment);
821             }
822             else
823             {
824                 float alt_g = temp->g_val + CalculateDirectDistance(temp->
                        GetPoint(), temp->FollowLink(i)->GetPoint(), true);
825                 if(temp->FollowLink(i)->g_val > alt_g)
826                 {
827                     float adj = alt_g - temp->FollowLink(i)->g_val;
828                     if(temp->predecessor == temp->FollowLink(i))
829                         int cat = 7;
830 
831                     temp->FollowLink(i)->predecessor = temp;
832                     AdjustSuccessors(temp->FollowLink(i), adj);
833                 }
834             }
835         }
836     }
837 
838     /** Last In First Out search*/
839     template <class NODE>
840     void DepthFirst(Robot* robot)
841     {
842         NODE *currentLocation, *goalLocation;
843         robot->GetMapNodes<NODE>(&currentLocation, &goalLocation);
844 
845         if(RecursDF(currentLocation, goalLocation))
846         {
847             int numNodes = 1;
848             NODE* tempNode = goalLocation;
849             Point* path;
850 
851             while(tempNode != currentLocation)
852             {
853                 tempNode = tempNode->predecessor;
854                 numNodes++;
855             }
856 
857             robot->SetNumSteps(numNodes);
858             path = (Point*) calloc(numNodes, sizeof(Point));
```

```
859             tempNode = goalLocation;
860             for( int i = numNodes - 1; i >= 0; i--)
861             {
862                 path[i] = tempNode->GetPoint();
863                 tempNode = tempNode->predecessor;
864             }
865             robot->SetPath(path);
866             free(path);
867             path = NULL;
868         }
869     }
870
871     template <class NODE>
872     bool RecursDF(NODE* node, NODE* goal)
873     {
874         bool result = false;
875         for(int i = 0; i < node->GetNumLinks(); i++)
876         {
877             NODE* temp = node->FollowLink(i);
878             if(!temp->CheckForMark())
879             {
880                 temp->MarkNode();
881                 if(temp == goal || RecursDF(temp, goal))
882                 {
883                     temp->predecessor = node;
884                     result = true;
885                 }
886             }
887         }
888         return result;
889     }
890
891     /** Simple planning algorithm which follows the boundaries of hazards
892         Have it check ahead of self, to allow for size of robot
893         Have it save end waypoint, though just checking ok to move forward 10 in
                that direction
894     */
895     template <class NODE>
896     void EightPointMove(Robot* robot)
897     {
898         float moveDist = MOVE_INCREMENTS_MM;
899         float dist = -1;
900         int ind = -1;
901         Point* path = (Point*) calloc(1, sizeof(Point));
902         Point tCurrentPos = robot->terrain->GetCurrentNode()->GetPoint();
903         Point dirVect = robot->GetGoalPt() - (tCurrentPos- Point(0,0,0));
```

```
904         Point moves[8] = {Point(moveDist,0,0), Point(moveDist,0,moveDist), Point(
                moveDist,0,-moveDist),
905                 Point(0,0,moveDist), Point(0,0,-moveDist),
906                 Point(-moveDist,0,-moveDist), Point(-moveDist,0,moveDist), Point
                        (-moveDist,0,0)};
907
908         for(int i = 0; i < 8; i++)
909         {
910             Point tNewPos = tCurrentPos + (moves[i] - Point(0,0,0));
911             Face_handle newTri;
912             if(robot->terrain->GetTri(&newTri, &tNewPos))
913             {
914                 if(newTri->CheckValidTri() && !newTri->hazardType)
915                 {
916                     //add check for clearance
917
918                     float tDist = CalculateDirectDistance(tCurrentPos, tNewPos,
                            true);
919                     if(dist < 0 || tDist < dist)
920                     {
921                         dist = tDist;
922                         ind = i;
923                     }
924                 }
925             }
926         }
927
928         if(ind != -1)
929             path[0] = tCurrentPos + (moves[ind] - Point(0,0,0));
930         else
931             path[0] = tCurrentPos;
932
933         robot->SetNumSteps(1);
934         robot->SetPath(path);
935     }
936
937     /** Simple state-model based algorithm, which cycles through the directions
            of forward, left, back and right.
938         Attempts to move until a hazard is encounter in which case next direction
                is tried,
939         Or if an obstacle has been successfully skirted then return to moving in
                previous direction */
940     template <class NODE>
941     void StateAlg(Robot* robot)
942     {
943         Point dirVect, normVect, frontLeft, frontRight, backRight, backLeft,
                currentPos;
```

```
944         Point* path = (Point*) calloc(1, sizeof(Point));
945         bool pathFound = false;
946         NODE *currentLocation, *goalLocation;
947         robot->GetMapNodes<NODE>(&currentLocation, &goalLocation);
948
949         currentPos = currentLocation->GetPoint();
950
951         //dirVect initially scaled to 10, to avoid overflow in cross product,
                then it and normVect are scaled to visible dist
952         dirVect = goalLocation->GetPoint() - (currentPos - ZERO_POINT);
953         dirVect =  NormaliseVector(dirVect, 10, false);
954         normVect = ScaledCrossProduct(dirVect, dirVect + (Point(0,0,10) -
                ZERO_POINT), SELECTION_RANGE);
955         dirVect =  NormaliseVector(dirVect, SELECTION_RANGE, false);
956
957         frontLeft = currentPos + (dirVect - normVect);
958         frontRight = currentPos + (dirVect - ZERO_POINT) + (normVect - ZERO_POINT
                );
959         backRight = currentPos + (normVect - dirVect);
960         backLeft = currentPos - (dirVect - normVect);// - ZERO_POINT);
961
962         path[0] = currentPos;
963         for(int i = 0; i < currentLocation->GetNumLinks() && !pathFound; i++)
964         {
965             NODE* temp = currentLocation->FollowLink(i);
966             switch(currentState)
967             {
968                 case FORWARD:
969                     if(InsideTriangleTest(temp->GetPoint(), currentPos, frontLeft
                            , frontRight))
970                     {
971                         if(currentState == FORWARD || (fwdSide == FWDRIGHT &&
                                currentState == RIGHT) || (fwdSide == FWDLEFT &&
                                currentState == LEFT))
972                         {
973                             currentState = FORWARD;
974                             path[0] = temp->GetPoint();
975                             pathFound = true;
976                             break;
977                         }
978                     }
979                 case RIGHT:
980                     if(InsideTriangleTest(temp->GetPoint(), currentPos,
                            frontRight, backRight))
981                     {
982                         if(currentState == RIGHT || (fwdSide == FWDRIGHT &&
                                currentState == BACK))
```

```
983                         {
984                             currentState = RIGHT;
985                             path[0] = temp->GetPoint();
986                             pathFound = true;
987                             break;
988                         }
989                     }
990                 case BACK:
991                     if(InsideTriangleTest(temp->GetPoint(), currentPos, backLeft,
                            backRight))
992                     {
993                         if(currentState == BACK || (fwdSide == FWDRIGHT &&
                                currentState == LEFT) || (fwdSide == FWDLEFT &&
                                currentState == RIGHT))
994                         {
995                             currentState = BACK;
996                             path[0] = temp->GetPoint();
997                             pathFound = true;
998                             break;
999                         }
1000                    }
1001                case LEFT:
1002                    if(InsideTriangleTest(temp->GetPoint(), currentPos, frontLeft
                            , frontRight))
1003                    {
1004                        if(currentState == LEFT || (fwdSide == FWDLEFT &&
                                currentState == BACK))
1005                        {
1006                            currentState = LEFT;
1007                            path[0] = temp->GetPoint();
1008                            pathFound = true;
1009                            break;
1010                        }
1011                    }
1012            }
1013        }
1014
1015        if(!pathFound)
1016        {
1017            if(fwdSide == FWDLEFT)
1018            {
1019                if(currentState == FORWARD)
1020                    currentState = LEFT;
1021                else if(currentState == LEFT)
1022                    currentState = BACK;
1023                else if(currentState == BACK)
1024                    currentState = RIGHT;
```

```
1025                else
1026                {
1027                    fwdSide = FWDRIGHT;
1028                    currentState = FORWARD;
1029                }
1030            }
1031            else
1032            {
1033                if(currentState == FORWARD)
1034                    currentState = RIGHT;
1035                else if(currentState == RIGHT)
1036                    currentState = BACK;
1037                else if(currentState == BACK)
1038                    currentState = LEFT;
1039                else
1040                {
1041                    fwdSide = FWDLEFT;
1042                    currentState = FORWARD;
1043                }
1044            }
1045        }
1046
1047        robot->SetNumSteps(1);
1048        robot->SetPath(path);
1049    }
1050
1051    void GreedyPaths(Robot* robot);
1052
1053 #endif
```

A.3 Display

A.3.1 Display.h

```
1  /*************************************************************************
2   *   Copyright (C) 2008 by Michael Douglas Hasler   *
3   *   Redheadpunk@gmail.com   *
4   *                                                                       *
5   *   This program is free software; you can redistribute it and/or modify  *
6   *   it under the terms of the GNU General Public License as published by  *
7   *   the Free Software Foundation; either version 2 of the License, or     *
8   *   (at your option) any later version.                                   *
9   *                                                                       *
10  *   This program is distributed in the hope that it will be useful,       *
11  *   but WITHOUT ANY WARRANTY; without even the implied warranty of        *
```

```
12  *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the          *
13  *   GNU General Public License for more details.                           *
14  *                                                                          *
15  *   You should have received a copy of the GNU General Public License      *
16  *   along with this program; if not, write to the                          *
17  *   Free Software Foundation, Inc.,                                        *
18  *   59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.              *
19  ****************************************************************************/
20
21  #if defined(_MSC_VER)
22  #include "SDL.h"
23  #include "SDL_gfxPrimitives.h"
24  #else
25  #include "SDL/SDL.h"
26  #include "SDL/SDL_gfxPrimitives.h"
27  #endif
28
29  enum MODES{
30      POINT_M = 1,
31      LINE_M,
32      TRI_M
33      };
34
35  int runDisplay(int* tris, int numTri, MODES drawMode, int* colourChangeIndex, int
        numChanges);
36  void render(int* tris, int numTri, MODES drawMode, int* colourChangeIndex, int
        numChanges);
37  int runTest();
38  void renderTest();
```

A.3.2 Display.cpp

```
1   /****************************************************************************
2    *   Copyright (C) 2008 by Michael Douglas Hasler   *
3    *   Redheadpunk@gmail.com   *
4    *                                                                          *
5    *   This program is free software; you can redistribute it and/or modify   *
6    *   it under the terms of the GNU General Public License as published by   *
7    *   the Free Software Foundation; either version 2 of the License, or      *
8    *   (at your option) any later version.                                    *
9    *                                                                          *
10   *   This program is distributed in the hope that it will be useful,        *
11   *   but WITHOUT ANY WARRANTY; without even the implied warranty of         *
12   *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the          *
13   *   GNU General Public License for more details.                           *
14   *                                                                          *
```

```
15  *    You should have received a copy of the GNU General Public License     *
16  *    along with this program; if not, write to the                         *
17  *    Free Software Foundation, Inc.,                                       *
18  *    59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.             *
19  ***************************************************************************/
20  #include "Display.h"
21
22  #define DEBUG true
23
24  const int WINDOW_WIDTH = 800;
25  const int WINDOW_HEIGHT = 800;
26  const char* WINDOW_TITLE = "SDL Start";
27  SDL_Surface *screen;
28
29  int runDisplay(int* tris, int numTriVertices, MODES drawMode, int*
        colourChangeIndex, int numChanges)
30  {
31      if(!DEBUG)
32          return 0;
33
34      if(SDL_Init(SDL_INIT_VIDEO) < 0)
35      {
36          fprintf(stderr, "Unable to init SDL: %s\n", SDL_GetError());
37          exit(1);
38      }
39
40      atexit(SDL_Quit);
41
42      screen = SDL_SetVideoMode( WINDOW_WIDTH, WINDOW_HEIGHT, 32, SDL_SWSURFACE);
43      SDL_WM_SetCaption( WINDOW_TITLE, 0 );
44
45      if(screen == NULL)
46      {
47          fprintf(stderr, "Unable to set 640x480 video: %s\n", SDL_GetError());
48          exit(1);
49      }
50
51      while(1)
52      {
53          SDL_Event event;
54          while(SDL_PollEvent(&event))
55          {
56              render(tris, numTriVertices, drawMode, colourChangeIndex, numChanges)
                    ;
57              switch(event.type)
58              {
59                  case SDL_KEYDOWN:
```

```
60                      break;
61                  case SDL_KEYUP:
62                      // If escape is pressed, return (and thus, quit)
63                      if(event.key.keysym.sym == SDLK_ESCAPE)
64                          return 0;
65                      break;
66                  case SDL_QUIT:
67                  {
68                      SDL_Quit();
69                      return ( 0 );
70                  }
71              }
72          }
73      }
74      return 0;
75  }
76
77  void render(int* tris, int numTriVertices, MODES drawMode, int* colourChangeIndex
        , int numChanges)
78  {
79      SDL_FillRect ( screen, NULL, SDL_MapRGB ( screen->format, 0, 0, 0 ) );
80      int rFill = 0, gFill = 255, bFill = 255, aFill = 255;
81      int rLine = 255, gLine = 0, bLine = 0, aLine = 255;
82      int nChange = 0;
83
84      switch(drawMode)
85      {
86          case TRI_M:
87          {
88              for(int i = 0; i < numTriVertices*2; i += 6)
89              {
90                  filledTrigonRGBA ( screen,
91                          tris[i], tris[i+1],
92                          tris[i+2], tris[i+3],
93                          tris[i+4], tris[i+5],
94                          rFill, (i*10)%gFill, bFill, aFill );
95              }
96
97              for(int i = 0; i < numTriVertices*2; i += 6)
98              {
99                  if(i == colourChangeIndex[nChange]*2)
100                 {
101                     rLine = 0;
102                     gLine = 255;
103                     nChange++;
104                 }
105                 trigonRGBA ( screen,
```

```
106                         tris[i], tris[i+1],
107                         tris[i+2], tris[i+3],
108                         tris[i+4], tris[i+5],
109                         rLine, gLine, bLine, aLine );
110             }
111             break;
112         }
113         case LINE_M:
114         {
115             for(int i = 0; i < numTriVertices*2; i += 4)
116             {
117                 if(i == colourChangeIndex[nChange]*2)
118                 {
119                     nChange++;
120                     rLine = 120;
121                     if(numChanges == 0)
122                         gLine = 120;
123                     else
124                         gLine = 255/numChanges * nChange;
125                     bLine = 200 * nChange%2;
126                 }
127 
128                 lineRGBA ( screen,
129                         tris[i], tris[i+1],
130                         tris[i+2], tris[i+3],
131                         rLine, gLine, bLine, aLine );
132             }
133             break;
134         }
135         case POINT_M:
136         {
137             for(int i = 0; i < numTriVertices*2; i += 2)
138             {
139                 if(i == colourChangeIndex[nChange]*2)
140                 {
141                     rLine = 0;
142                     gLine = 255;
143                     nChange++;
144                 }
145                 pixelRGBA ( screen,
146                         tris[i], tris[i+1],
147                         rLine, gLine, bLine, aLine );
148             }
149             break;
150         }
151     }
152 
```

```
153     SDL_Flip ( screen );
154 }
155
156 int runTest()
157 {
158     // Initialize SDL's subsystems - in this case, only video.
159     if(SDL_Init(SDL_INIT_VIDEO) < 0)
160     {
161         fprintf(stderr, "Unable to init SDL: %s\n", SDL_GetError());
162         exit(1);
163     }
164
165     // Register SDL_Quit to be called at exit; makes sure things are
166     // cleaned up when we quit.
167     atexit(SDL_Quit);
168
169     // Attempt to create a 640x480 window with 32bit pixels.
170     screen = SDL_SetVideoMode( WINDOW_WIDTH, WINDOW_HEIGHT, 32, SDL_SWSURFACE);
171     //SDL_Surface* screen = SDL_SetVideoMode( WINDOW_WIDTH, WINDOW_HEIGHT, 0,
            SDL_HWSURFACE | SDL_DOUBLEBUF );
172
173     SDL_WM_SetCaption( WINDOW_TITLE, 0 );
174
175     // If we fail, return error.
176     if(screen == NULL)
177     {
178         fprintf(stderr, "Unable to set 640x480 video: %s\n", SDL_GetError());
179         exit(1);
180     }
181
182     while(1)
183     {
184         renderTest();
185         // Poll for events, and handle the ones we care about.
186         SDL_Event event;
187         while(SDL_PollEvent(&event))
188         {
189             switch(event.type)
190             {
191                 case SDL_KEYDOWN:
192                     break;
193                 case SDL_KEYUP:
194                     // If escape is pressed, return (and thus, quit)
195                     if(event.key.keysym.sym == SDLK_ESCAPE)
196                         return 0;
197                     break;
198                 case SDL_QUIT:
```

```
199                 {
200                     SDL_Quit();
201                     return ( 0 );
202                 }
203             }
204         }
205     }
206     return 0;
207 }
208
209 void renderTest()
210 {
211     int tick = SDL_GetTicks();
212     SDL_FillRect ( screen, NULL, SDL_MapRGB ( screen->format, 0, 0, 0 ) );
213     trigonRGBA ( screen,
214                  500, 50,
215                  550, 200,
216                  600, 150,
217                  0, 255, 255, 255 );
218
219     filledTrigonRGBA ( screen,
220                        200 + (tick/100) % 50, 200 - (tick/100) % 25,
221                        300, 50,
222                        400, 200,
223                        0, 0, 255, 255 );
224
225     SDL_Flip ( screen );
226 }
```

A.4 Hier_Triangulation

A.4.1 Hier_Triangulation.h

```
1  /*****************************************************************************
2   *   Copyright (C) 2007 by Michael Douglas Hasler    *
3   *   Redheadpunk@gmail.com    *
4   *                                                                           *
5   *   This program is free software; you can redistribute it and/or modify    *
6   *   it under the terms of the GNU General Public License as published by    *
7   *   the Free Software Foundation; either version 2 of the License, or       *
8   *   (at your option) any later version.                                     *
9   *                                                                           *
10  *   This program is distributed in the hope that it will be useful,         *
11  *   but WITHOUT ANY WARRANTY; without even the implied warranty of          *
12  *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the           *
```

```
13  *   GNU General Public License for more details.                          *
14  *                                                                         *
15  *   You should have received a copy of the GNU General Public License     *
16  *   along with this program; if not, write to the                         *
17  *   Free Software Foundation, Inc.,                                       *
18  *   59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.             *
19  *************************************************************************/
20 #ifndef TRIANGULATION_H_
21 #define TRIANGULATION_H_
22
23 #include <fstream>
24 #include  <cassert>
25 #include  <list>
26 #include  <vector>
27
28 #include <CGAL/basic.h>
29 #include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
30
31 #include "adv_math.h"
32 #include "Projection_traits_xy_3.h"
33 #include <CGAL/intersections.h>
34 #include <CGAL/Delaunay_triangulation_2.h>
35 #include <CGAL/Triangulation_hierarchy_2.h>
36
37 struct K : CGAL::Exact_predicates_inexact_constructions_kernel {};
38 typedef CGAL::Projection_traits_xy_3<K>              Et;
39 typedef CGAL::Triangulation_vertex_base_2<Et>            Vb2;
40 typedef CGAL::Triangulation_hierarchy_vertex_base_2<Vb2>     Vbh2;
41
42 template < class Gt, class Fb = CGAL::Triangulation_face_base_2<Gt> >
43 class Altered_triangulation_face_2 : public  Fb{
44     typedef Fb Base;
45 public:
46     typedef typename Fb::Vertex_handle                    Vertex_handle;
47     typedef typename Fb::Face_handle                      Face_handle;
48     typedef K::Point_3    Point;
49
50     template < typename TDS2 >
51     struct Rebind_TDS {
52         typedef typename Fb::template Rebind_TDS<TDS2>::Other Fb2;
53         typedef Altered_triangulation_face_2<Gt,Fb2> Other;
54     };
55
56     Altered_triangulation_face_2() : Base()
57     {
58         hazardType = NONE; checked = false; newlyCreated = true;
59     }
```

```
60      Altered_triangulation_face_2(Vertex_handle v0, Vertex_handle v1,
            Vertex_handle v2) : Base(v0,v1,v2)
61      {
62          hazardType = NONE; checked = false; newlyCreated = true;
63      }
64      Altered_triangulation_face_2(Vertex_handle v0, Vertex_handle v1,
            Vertex_handle v2, Face_handle n0, Face_handle n1, Face_handle n2) : Base(
            v0,v1,v2,n0,n1,n2)
65      {
66          hazardType = NONE; checked = false; newlyCreated = true;
67      }
68
69      bool CheckValidTri()
70      {
71          if(this != NULL && hazardType < MAX_HAZARD && hazardType >= MIN_HAZARD &&
                GetPoint(0) != NULL && GetPoint(1) != NULL && GetPoint(2) != NULL)
72              return true;
73          else
74              return false;
75      }
76
77      //use typeinfo package to test hazardType is of type HAZARDS ie within bounds
78      Point* GetPoint(int num)
79      {
80          if(num < 3 && num >=0 && typeid(this->vertex(num)) == typeid(
                Vertex_handle) && typeid(this->vertex(num)->point()) == typeid(Point)
                )
81              return (Point*) &(this->vertex(num)->point());
82          else
83              return NULL;
84      }
85
86      HAZARDS  hazardType;
87      bool checked;
88      bool newlyCreated;
89  };
90
91  //typedef CGAL::Triangulation_face_base_2<Et>           Fb2;
92  typedef Altered_triangulation_face_2<Et>              Fb2;
93  typedef CGAL::Triangulation_data_structure_2<Vbh2,Fb2>      Tds2;
94  typedef CGAL::Delaunay_triangulation_2<Et,Tds2>       Delaunay2;
95  typedef CGAL::Triangulation_hierarchy_2<Delaunay2>    TriangulationHier2;
96
97  class Hierarchy2DMesh;
98
99  class Hierarchy2DMesh{
100 public:
```

```
101     typedef K::Point_3    Point;
102     typedef K::Vector_3    Vector;
103     typedef TriangulationHier2::Locate_type Locate_type;
104     typedef TriangulationHier2::Face_handle Face_handle;
105     typedef TriangulationHier2::Face_iterator Face_iterator;
106     typedef TriangulationHier2::Vertex_handle Vertex_handle;
107     typedef TriangulationHier2::Line_face_circulator Line_face_circulator;
108
109 //  typedef TriangulationHier2::Triangle_2<K> Tri_2;
110 //  typedef TriangulationHier2::Triangle_3 Tri_3;
111
112     struct mesh_Triangle : Triangle
113     {
114         mesh_Triangle(Coord* pointA, Coord* pointB, Coord* pointC, Face_handle
                hTri) : Triangle(pointA, pointB, pointC){
115             assoc_hierTri = hTri;
116             hazardType = hTri->hazardType;
117         };
118         Face_handle assoc_hierTri;
119     };
120
121     Hierarchy2DMesh();
122     Hierarchy2DMesh(float* coords, int numCoords);
123     void AddPoint(float x, float y, float z);
124     void AddPoints(float* coords, int numCoords);
125     void RemoveVert(Point oldPt);
126     bool FindTri(float x, float y, float z,  Triangle* tri);
127     bool FindTri(float x, float y, float z,  Face_handle* tri);
128     void GetTrianglesIter(Triangle*** triangles, int* numTriangles);
129     void GetTrianglesIter(Face_handle** triangles, int* numTriangles);
130     void GetTerrainNodes(Coord** terrNodes, int* numTerrNodes);
131     bool CheckBetweenPoints();
132 //  bool CheckIfInfinite(Face_handle f);
133 //  bool CheckIfInfinite(Vertex_handle v);
134
135     template <class OBJ>
136     bool CheckIfInfinite(OBJ object){
137         return mesh.is_infinite(object);
138     }
139
140
141     bool CheckLineForHazards(Point start, Point end, bool checkParallel, Point
            normVector, bool checkForSlope, bool *isSlope);
142
143     void Assign(Face_handle &face, Face_handle &values)
144     {
145         if (&face != &values)
```

```
146         {
147             face = values;
148             face->hazardType = values->hazardType; face->checked = values->
                    checked;
149         }
150     }
151
152 private:
153     TriangulationHier2 mesh;
154
155     //mesh->clear() use when dis-orientated etc
156 };
157 #endif
```

A.4.2 Hier_Triangulation.cpp

```
1  /*************************************************************************
2   *   Copyright (C) 2007 by Michael Douglas Hasler   *
3   *   Redheadpunk@gmail.com   *
4   *                                                                       *
5   *   This program is free software; you can redistribute it and/or modify  *
6   *   it under the terms of the GNU General Public License as published by  *
7   *   the Free Software Foundation; either version 2 of the License, or     *
8   *   (at your option) any later version.                                   *
9   *                                                                         *
10  *   This program is distributed in the hope that it will be useful,       *
11  *   but WITHOUT ANY WARRANTY; without even the implied warranty of        *
12  *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the         *
13  *   GNU General Public License for more details.                          *
14  *                                                                         *
15  *   You should have received a copy of the GNU General Public License     *
16  *   along with this program; if not, write to the                         *
17  *   Free Software Foundation, Inc.,                                       *
18  *   59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.             *
19  *************************************************************************/
20 #include "adv_math.h"
21 #include "Robot.h"
22 #include "Hier_Triangulation.h"
23
24     Hierarchy2DMesh::Hierarchy2DMesh()
25     {
26         mesh = TriangulationHier2();
27     }
28
29     Hierarchy2DMesh::Hierarchy2DMesh(float* coords, int arraySize)
30     {
```

```
31          std::list<Point> coordList;
32          for(int i = 0; i < arraySize; i += 3)
33          {
34              coordList.push_front(Point(coords[i], coords[i+1], coords[i+2]));
35          }
36
37          mesh = TriangulationHier2();
38          mesh.insert(coordList.begin(), coordList.end());
39      }
40
41      void Hierarchy2DMesh::AddPoint(float x, float y, float z)
42      {
43          mesh.insert(Point(x,y,z));
44      }
45
46      void Hierarchy2DMesh::AddPoints(float* coords, int arraySize)
47      {
48          std::list<Point> coordList;
49          for(int i = 0; i < arraySize; i += 3)
50              coordList.push_front(Point(coords[i], coords[i+1], coords[i+2]));
51          mesh.insert(coordList.begin(), coordList.end());
52      }
53
54      void Hierarchy2DMesh::RemoveVert(Point oldPt)
55      {
56          bool result = false;
57          Locate_type l_type = Delaunay2::VERTEX;
58          int l_index = 0;
59
60          Face_handle face = mesh.locate(oldPt, l_type, l_index);
61          if(l_type == Delaunay2::VERTEX )
62          {
63              Vertex_handle oldVert = face->vertex(l_index);
64              mesh.remove(oldVert);
65          }
66          else //if(l_type == Delaunay2::EDGE  l_type == Delaunay2::FACE))
67          {
68              printf("Error - Point to remove was not found to be a vertex\n");
69          }
70      }
71
72      /** Only called by AddPoint via GetTri() */
73      bool Hierarchy2DMesh::FindTri(float x, float y, float z, Triangle* tri)
74      {
75          bool result = false;
76          Locate_type l_type = Delaunay2::VERTEX;
77          int l_index = 0;
```

```
78          Point pt(x,y,z);
79
80          Face_handle face = mesh.locate(pt, l_type, l_index);
81          if( l_type == Delaunay2::FACE || l_type == Delaunay2::VERTEX || l_type ==
                Delaunay2::EDGE) //Edge
82          {
83              result = true;
84              tri->point1->xVal = face->vertex(0)->point().x();
85              tri->point1->yVal = face->vertex(0)->point().y();
86              tri->point1->zVal = face->vertex(0)->point().z();
87
88              tri->point2->xVal = face->vertex(1)->point().x();
89              tri->point2->yVal = face->vertex(1)->point().y();
90              tri->point2->zVal = face->vertex(1)->point().z();
91
92              tri->point3->xVal = face->vertex(2)->point().x();
93              tri->point3->yVal = face->vertex(2)->point().y();
94              tri->point3->zVal = face->vertex(2)->point().z();
95
96              tri->checked = false;
97              tri->hazardType = face->hazardType;
98          }
99          else if( l_type == Delaunay2::OUTSIDE_CONVEX_HULL)
100         {
101             int a = 0, b = 1;
102             if(CheckIfInfinite(face->vertex(1)))
103                 b = 2;
104             else if(CheckIfInfinite(face->vertex(0)))
105                 a = 2;
106
107             if(face->vertex(a) != NULL)
108             {
109                 tri->point1->xVal = face->vertex(a)->point().x();
110                 tri->point1->yVal = face->vertex(a)->point().y();
111                 tri->point1->zVal = face->vertex(a)->point().z();
112             }
113
114             if(face->vertex(b) != NULL)
115             {
116                 tri->point2->xVal = face->vertex(b)->point().x();
117                 tri->point2->yVal = face->vertex(b)->point().y();
118                 tri->point2->zVal = face->vertex(b)->point().z();
119             }
120         }
121
122         return result;
123     }
```

```
124
125     bool Hierarchy2DMesh::FindTri(float x, float y, float z, Face_handle* tri)
126     {
127         bool result = false;
128         Locate_type l_type = Delaunay2::VERTEX;
129         int l_index = 0;
130         Point pt(x,y,z);
131
132         Face_handle face = mesh.locate(pt, l_type, l_index);
133         if( l_type == Delaunay2::FACE || l_type == Delaunay2::VERTEX )
134         {
135             result = true;
136             *tri = face;
137         }
138         else if(l_type == Delaunay2::EDGE)
139         {
140             result = true;
141             *tri = face;
142         }
143         else if(l_type == Delaunay2::OUTSIDE_CONVEX_HULL) //So can test inf tri
                against each other
144             *tri = face;
145
146         return result;
147     }
148
149     void Hierarchy2DMesh::GetTrianglesIter(Triangle*** triangles, int*
            numTriangles)
150     {
151         Face_iterator fIterator = mesh.faces_begin();
152         Face_iterator theEnd = mesh.faces_end ();
153
154         while( fIterator != theEnd)
155         {
156             Face_handle face = fIterator;
157
158             Point pt0 = face->vertex(0)->point();
159             Point pt1 = face->vertex(1)->point();
160             Point pt2 = face->vertex(2)->point();
161
162             (*triangles)[(*numTriangles)++] = (Triangle*) new mesh_Triangle(new
                    Coord(pt0.x(),pt0.y(),pt0.z()), new Coord(pt1.x(),pt1.y(),pt1.z()
                    ), new Coord(pt2.x(),pt2.y(),pt2.z()), face);
163
164             if(*numTriangles % ARRAY_CHUNK == 0)
165             {
```

```
166                 void* tmp = realloc(*triangles, (*numTriangles + ARRAY_CHUNK)*
                        sizeof(Triangle*));
167                 *triangles = (Triangle**) tmp;
168             }
169
170             fIterator++;
171         }
172
173         return;
174     }
175
176     void Hierarchy2DMesh::GetTrianglesIter(Face_handle** triangles, int*
        numTriangles)
177     {
178         Face_iterator fIterator = mesh.faces_begin();
179         Face_iterator theEnd = mesh.faces_end();
180         if(!mesh.is_valid())
181         {
182             int cat = 7;
183             printf("Error - Triangulation has become invalid");
184             fIterator = theEnd;
185         }
186
187         while( fIterator != theEnd)
188         {
189             Face_handle face = fIterator;
190             if(face->CheckValidTri())
191                 (*triangles)[(*numTriangles)++] = face;
192             else
193             {
194                 int cat = 7;
195                 printf("Error - Face in Triangulation was invalid");
196             }
197
198
199             if(*numTriangles % ARRAY_CHUNK == 0)
200             {
201                 void* tmp = realloc(*triangles, (*numTriangles + ARRAY_CHUNK)*
                        sizeof(Face_handle));
202                 *triangles = (Face_handle*) tmp;
203             }
204
205             fIterator++;
206         }
207
208         return;
209     }
```

```
210
211     /** Iterates through triangles, finding hazards bordering non-hazard tris,
            then creates points clear of the hazard
212
213         - Possibly change from finding hazards, to finding nonhazards with a
                hazard neighbour, which is more intensive?
214         - Insteading of adding vector off hazard edge, maybe take nonhazard and
                going of it/centre or something
215      */
216     void Hierarchy2DMesh::GetTerrainNodes(Coord** terrNodes, int* numTerrNodes)
217     {
218         Face_iterator fIterator = mesh.faces_begin();
219         Face_iterator theEnd = mesh.faces_end();
220
221         while( fIterator != theEnd)
222         {
223             Face_handle face = fIterator;
224             if(face->hazardType)
225             {
226                 if(!face->neighbor(0)->hazardType)
227                 {
228                     Coord pointA = ConvertPointType<TriangulationHier2::Point,
                            Coord>(*face->GetPoint(1));
229                     Coord pointB = ConvertPointType<TriangulationHier2::Point,
                            Coord>(*face->GetPoint(2));
230                     Coord vectorAB = pointB - pointA;
231                     Coord normAB = NormaliseVector(CrossProduct(vectorAB,
                            vectorAB + Coord(0,0,10)), CLEARANCEBUFFER_MM, false);
232                     (*terrNodes)[(*numTerrNodes)++] = pointA + normAB;
233                     (*terrNodes)[(*numTerrNodes)++] = pointB + normAB;
234                 }
235
236                 if(!face->neighbor(1)->hazardType)
237                 {
238                     Coord pointA = ConvertPointType<TriangulationHier2::Point,
                            Coord>(*face->GetPoint(0));
239                     Coord pointB = ConvertPointType<TriangulationHier2::Point,
                            Coord>(*face->GetPoint(2));
240                     Coord vectorAB = pointB - pointA;
241                     Coord normAB = NormaliseVector(CrossProduct(vectorAB,
                            vectorAB + Coord(0,0,10)), CLEARANCEBUFFER_MM, false);
242                     (*terrNodes)[(*numTerrNodes)++] = pointA + normAB;
243                     (*terrNodes)[(*numTerrNodes)++] = pointB + normAB;
244                 }
245
246                 if(!face->neighbor(2)->hazardType)
247                 {
```

```
248                     Coord pointA = ConvertPointType<TriangulationHier2::Point,
                            Coord>(*face->GetPoint(0));
249                     Coord pointB = ConvertPointType<TriangulationHier2::Point,
                            Coord>(*face->GetPoint(1));
250                     Coord vectorAB = pointB - pointA;
251                     Coord normAB = NormaliseVector(CrossProduct(vectorAB,
                            vectorAB + Coord(0,0,10)), CLEARANCEBUFFER_MM, false);
252                     (*terrNodes)[(*numTerrNodes)++] = pointA + normAB;
253                     (*terrNodes)[(*numTerrNodes)++] = pointB + normAB;
254                 }
255             }
256
257             if(*numTerrNodes % ARRAY_CHUNK == 0)
258             {
259                 void* tmp = realloc(*terrNodes, (*numTerrNodes + ARRAY_CHUNK)*
                        sizeof(Coord));
260                 *terrNodes = (Coord*) tmp;
261             }
262
263             fIterator++;
264         }
265
266         return;
267
268     }
269
270     //Maybe inverse result or change name as they seem at odds
271     /** Returns false if a hazard is encountered. Uses two line_face_circulators
            (reverse one needed for faces bordering line but lying to the left, refer
             CGAL manual) to get all faces intersected by a line, then cycles through
             so only the ones between start and end are checked, then looks whether
            they are hazards. Faces which are intersected at vertex are omitted, thus
             the lines are offset to overlap coverage slightly to ensure all
            triangles are checked.
272
273         Infinite check is for when reaches end of faces, to avoid looping through
                again.
274         isSlope is used to signify if is traversable in one direction. Requires
                that no other hazard types be encountered.
275     */
276     bool Hierarchy2DMesh::CheckLineForHazards(Point start, Point end, bool
            checkParallel, Point normVect, bool checkForSlope, bool *isSlope)
277     {
278         Vector pOffset = NormaliseVector(normVect, 2.0f, false) - ZERO_POINT;
279
280         Line_face_circulator fCirclator = mesh.line_walk(start + pOffset, end +
                pOffset);
```

```
281         Line_face_circulator reverseCirclator = mesh.line_walk(end - pOffset,
                start - pOffset);
282         Face_handle begin = fCirclator;
283         Face_handle stop = reverseCirclator;
284         bool result = true;
285         int fLoops = 1;
286         int rLoops = 1;
287
288         while(fCirclator != NULL && fCirclator != stop)
289         {
290             Face_handle face = fCirclator;
291             if(CheckIfInfinite<Face_handle>(face) || InsideTriangleTest(start, *
                    face->GetPoint(0), *face->GetPoint(1), *face->GetPoint(2)))
292                 break;
293             fCirclator++;
294             fLoops++;
295         }
296         if(fCirclator == stop)
297             fCirclator = mesh.line_walk(start + pOffset, end + pOffset);
298
299         while(reverseCirclator != NULL && reverseCirclator != begin)
300         {
301             Face_handle face = reverseCirclator;
302             Point p0 = *face->GetPoint(0);
303             Point p1 = *face->GetPoint(1);
304             Point p2 = *face->GetPoint(2);
305             if(CheckIfInfinite<Face_handle>(face) || InsideTriangleTest(end, *
                    face->GetPoint(0), *face->GetPoint(1), *face->GetPoint(2)))
306                 break;
307             reverseCirclator++;
308             rLoops++;
309         }
310         if(reverseCirclator == begin)
311             reverseCirclator = mesh.line_walk(end - pOffset, start - pOffset);
312
313         begin = fCirclator;
314         stop = reverseCirclator;
315
316         if(begin != NULL)
317         {
318             Face_handle face = fCirclator;
319             while(result && fCirclator != stop && !CheckIfInfinite<Face_handle>(
                    face))
320 //              for(int i = 0; i < fLoops; i++)
321             {
322                 face = fCirclator++;
323                 Point p0 = *face->GetPoint(0);
```

```
324                 Point p1 = *face->GetPoint(1);
325                 Point p2 = *face->GetPoint(2);
326                 if(checkForSlope && (face->hazardType == SLOPE || face->
                        hazardType == PART_SLOPE))
327                     *isSlope = true;
328                 else if(face->hazardType)
329                     result = false;
330                 if(fCirclator == begin)//has looped through all faces
331                     break;
332             }
333         }
334 
335         if(result && stop != NULL)
336         {
337             while(result && reverseCirclator != begin && !CheckIfInfinite<
                        Face_handle>((Face_handle) reverseCirclator))
338 //          for(int i = 0; i < rLoops; i++)
339             {
340                 Face_handle face = reverseCirclator++;
341                 if(checkForSlope && (face->hazardType == SLOPE || face->
                        hazardType == PART_SLOPE))
342                     *isSlope = true;
343                 else if(face->hazardType)
344                     result = false;
345                 if(reverseCirclator == stop)//has looped through all faces
346                     break;
347             }
348         }
349 
350         if(result && checkParallel)
351         {
352             if( !CheckLineForHazards(start + (normVect - ZERO_POINT), end  + (
                        normVect - ZERO_POINT), false, normVect, checkForSlope, isSlope)
                        || !CheckLineForHazards(start - (normVect - ZERO_POINT), end  - (
                        normVect - ZERO_POINT), false, normVect, checkForSlope, isSlope)
                        )
353                 result = false;
354         }
355 
356         if(checkForSlope && *isSlope)
357         {
358             if(result)
359                 result = false;
360             else
361                 *isSlope = false;
362         }
363 
```

```
364         return result;
365     }
366
367     bool Hierarchy2DMesh::CheckBetweenPoints()
368     {
369     }
```

A.5 Nodes

A.5.1 Nodes.h

```
1  /*************************************************************************
2   *   Copyright (C) 2009 by Michael Douglas Hasler   *
3   *   redheadpunk@gmail.com   *
4   *                                                                       *
5   *   This program is free software; you can redistribute it and/or modify *
6   *   it under the terms of the GNU General Public License as published by *
7   *   the Free Software Foundation; either version 2 of the License, or    *
8   *   (at your option) any later version.                                  *
9   *                                                                       *
10  *   This program is distributed in the hope that it will be useful,      *
11  *   but WITHOUT ANY WARRANTY; without even the implied warranty of       *
12  *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the        *
13  *   GNU General Public License for more details.                         *
14  *                                                                       *
15  *   You should have received a copy of the GNU General Public License    *
16  *   along with this program; if not, write to the                        *
17  *   Free Software Foundation, Inc.,                                      *
18  *   59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.            *
19  *************************************************************************/
20 #ifndef NODES_H_
21 #define NODES_H_
22
23 #include "Hier_Triangulation.h"
24 typedef Hierarchy2DMesh::Face_handle     Face_handle;
25
26 const int LIST_CHUNK = 10;
27 const bool SHOW_NODE_ERRORS = true;
28 const bool SHOW_NODE_WARNINGS = false;
29
30 /**
31 A base Node class from which different ones extend
32     - Base class having, an xyz coord and equality functions
33 */
34 class PtNode{
```

```
35 public:
36     ~PtNode();
37     PtNode();
38     PtNode(int x, int y, int z);
39     PtNode(Point xyz);
40     Point GetPoint();
41     void Adjust(int x, int y, int z);
42     //virtual void SetPoint(Point xyz);
43     bool operator==(PtNode* node);
44     bool operator==(Point otherPoint);
45 protected:
46
47     Point XYZcoordinates;
48 };
49
50 /**
51  * Nodes of points bordering hazard tri
52  */
53 class HazardNode : public PtNode {
54 public:
55     HazardNode();
56     ~HazardNode();
57     HazardNode(Point xyz, Face_handle tri, int vertNumA, int vertNumB);
58     bool CheckValid();
59     Face_handle GetHazardTri();
60
61     int vertA, vertB;
62 private:
63     Face_handle hazardTri;
64 };
65
66 /**
67  * Create nodes and link or create then have function to check all combos and see
        if clear path between nodes
68  * For use with A* type algorithms
69  */
70 class MapNode : public PtNode {
71 public:
72     ~MapNode();
73     MapNode();
74     MapNode(int x, int y, int z);
75     MapNode(Point xyz);
76     MapNode(int x, int y, int z, Face_handle nodeA, Face_handle nodeB);
77     MapNode(Point xyz, Face_handle nodeA, Face_handle nodeB);
78     MapNode(int x, int y, int z, int numNeighbours, MapNode** neighbours);
79     MapNode(Point xyz, int numNeighbours, MapNode** neighbours);
80
```

```
81      int GetNumLinks();
82      void AddLink(MapNode* node);
83      void BreakLink(MapNode* node);
84      MapNode* FollowLink(int num);
85      bool CheckForMark();
86      void MarkNode();
87      Face_handle GetHazardTriA();
88      Face_handle GetHazardTriB();
89
90      MapNode* predecessor;
91  protected:
92      int numLinks;
93      Face_handle hazardA, hazardB;
94  private:
95      bool marked;
96      MapNode** linkedNodes;
97  };
98
99  /**
100  * Create nodes and link or create then have function to check all combos and see
         if clear path between nodes
101  * For use with A* type algorithms
102  */
103 class GridNode : public PtNode {
104 public:
105     ~GridNode();
106     GridNode();
107     GridNode(int x, int y, int z);
108     GridNode(Point xyz);
109     GridNode(int x, int y, int z, HAZARDS hazardType, bool clearance);
110     GridNode(Point xyz, HAZARDS hazardType, bool clearance);
111     GridNode(int x, int y, int z, int numNeighbours, GridNode** neighbours);
112     GridNode(Point xyz, int numNeighbours, GridNode** neighbours);
113
114     int GetNumLinks();
115     void AddLink(GridNode* node);
116     void BreakLink(GridNode* node);
117     void ExpandLinks(int direction, int stepSize, GridNode prevNode);
118     GridNode* FollowLink(int num);
119     bool CheckForMark();
120     bool CheckForClearance();
121     void MarkNode();
122
123     GridNode* predecessor;
124     HAZARDS hType;
125 protected:
126     int numLinks;
```

```
127 private:
128     bool marked;
129     bool clearance;
130     GridNode** linkedNodes;
131 };
132 
133 /**
134 
135  * For use with A* type algorithms based on graph of mapNodes formed from hNodes
136  */
137 class ASTAR_MapNode : public MapNode {
138 public:
139     ASTAR_MapNode();
140     ~ASTAR_MapNode();
141     ASTAR_MapNode(int x, int y, int z);
142     ASTAR_MapNode(Point xyz);
143     ASTAR_MapNode(int x, int y, int z, Face_handle nodeA, Face_handle nodeB);
144     ASTAR_MapNode(Point xyz, Face_handle nodeA, Face_handle nodeB);
145 
146     void AddLink(ASTAR_MapNode* node);
147     void BreakLink(ASTAR_MapNode* node);
148     ASTAR_MapNode* FollowLink(int num);
149     void ResetOpenClosed();
150 
151     bool open, closed;
152     float f_val, g_val, h_val;
153     ASTAR_MapNode* predecessor;
154     ASTAR_MapNode* successor;
155 private:
156     ASTAR_MapNode** linkedNodes;
157 };
158 
159 /**
160 
161  * For use with A* type algorithms based on gridNodes
162  */
163 class ASTAR_GridNode : public GridNode {
164 public:
165     ASTAR_GridNode();
166     ~ASTAR_GridNode();
167     ASTAR_GridNode(int x, int y, int z);
168     ASTAR_GridNode(Point xyz);
169     ASTAR_GridNode(int x, int y, int z, HAZARDS hazardType, bool clearance);
170     ASTAR_GridNode(Point xyz, HAZARDS hazardType, bool clearance);
171 
172     void AddLink(ASTAR_GridNode* node);
173     void BreakLink(ASTAR_GridNode* node);
```

```
174     void ExpandLinks(int direction, int stepSize, ASTAR_GridNode prevNode);
175     ASTAR_GridNode* FollowLink(int num);
176     void ResetOpenClosed();
177
178     bool open, closed;
179     float f_val, g_val, h_val;
180     ASTAR_GridNode* predecessor;
181     ASTAR_GridNode* successor;
182 private:
183     ASTAR_GridNode** linkedNodes;
184 };
185 #endif
```

A.5.2 Nodes.cpp

```
1  /*************************************************************************
2   *   Copyright (C) 2009 by Michael Douglas Hasler   *
3   *   redheadpunk@gmail.com   *
4   *                                                                       *
5   *   This program is free software; you can redistribute it and/or modify *
6   *   it under the terms of the GNU General Public License as published by *
7   *   the Free Software Foundation; either version 2 of the License, or    *
8   *   (at your option) any later version.                                  *
9   *                                                                       *
10  *   This program is distributed in the hope that it will be useful,      *
11  *   but WITHOUT ANY WARRANTY; without even the implied warranty of       *
12  *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the        *
13  *   GNU General Public License for more details.                         *
14  *                                                                       *
15  *   You should have received a copy of the GNU General Public License    *
16  *   along with this program; if not, write to the                        *
17  *   Free Software Foundation, Inc.,                                      *
18  *   59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.            *
19  *************************************************************************/
20 #include "Nodes.h"
21
22     /************************************************ PtNode Object
           ************************************************/
23
24     PtNode::PtNode()
25     {
26         XYZcoordinates = Point();
27     }
28
29     PtNode::PtNode(int x, int y, int z)
30     {
```

```
31          XYZcoordinates = Point(x,y,z);
32      }
33
34      PtNode::PtNode(Point xyz)
35      {
36          XYZcoordinates = Point(xyz.x(), xyz.y(), xyz.z());
37      }
38
39      PtNode::~PtNode()
40      {
41      }
42
43      Point PtNode::GetPoint()
44      {
45          return XYZcoordinates;
46      }
47
48      void PtNode::Adjust(int x, int y, int z)
49      {
50          XYZcoordinates += Point(x,y,z);
51      }
52
53      bool PtNode::operator==(PtNode* node)
54      {
55          bool result = false;
56          if(XYZcoordinates == node->GetPoint())
57              result = true;
58          return result;
59      }
60
61      bool PtNode::operator==(Point otherPoint)
62      {
63          bool result = false;
64          if(XYZcoordinates == otherPoint)
65              result = true;
66          return result;
67      }
68
69      /********************************************** HazardNode Object
            **********************************************/
70
71      HazardNode::HazardNode() : PtNode ()
72      {
73          hazardTri = NULL;
74          vertA = 0;
75          vertB = 0;
76      }
```

```
77
78      HazardNode::HazardNode(Point xyz, Face_handle tri, int vertNumA, int vertNumB
            ) : PtNode (xyz)
79      {
80          hazardTri = tri;
81          vertA = vertNumA;
82          vertB = vertNumB;
83      }
84
85      Face_handle HazardNode::GetHazardTri()
86      {
87          return hazardTri;
88      }
89
90      bool HazardNode::CheckValid()
91      {
92          bool result = false;
93          if(this != NULL && vertA > -1 && vertA < 3 && vertB > -1 && vertB < 3 )
94              result = true;
95          return result;
96      }
97
98      HazardNode::~HazardNode()
99      {
100         hazardTri = NULL;
101         vertA = -1;
102         vertB = -1;
103     }
104
105     /************************************************ MapNode Object
            ************************************************/
106
107     MapNode::MapNode() : PtNode()
108     {
109         predecessor = NULL;
110         hazardA = NULL;
111         hazardB = NULL;
112         marked = false;
113         numLinks = 0;
114         linkedNodes = (MapNode**) calloc(LIST_CHUNK, sizeof(MapNode*));
115     }
116
117     MapNode::MapNode(int x, int y, int z) : PtNode(x,y,z)
118     {
119         predecessor = NULL;
120         hazardA = NULL;
121         hazardB = NULL;
```

```
122         marked = false;
123         numLinks = 0;
124         linkedNodes = (MapNode**) calloc(LIST_CHUNK, sizeof(MapNode*));
125     }
126
127     MapNode::MapNode(Point xyz) : PtNode(xyz)
128     {
129         predecessor = NULL;
130         hazardA = NULL;
131         hazardB = NULL;
132         marked = false;
133         numLinks = 0;
134         linkedNodes = (MapNode**) calloc(LIST_CHUNK, sizeof(MapNode*));
135     }
136
137     MapNode::MapNode(int x, int y, int z, Face_handle nodeA, Face_handle nodeB) :
            PtNode(x,y,z)
138     {
139         predecessor = NULL;
140         hazardA = nodeA;
141         hazardB = nodeB;
142         marked = false;
143         numLinks = 0;
144         linkedNodes = (MapNode**) calloc(LIST_CHUNK, sizeof(MapNode*));
145     }
146
147     MapNode::MapNode(Point xyz, Face_handle nodeA, Face_handle nodeB) : PtNode(
            xyz)
148     {
149         predecessor = NULL;
150         hazardA = nodeA;
151         hazardB = nodeB;
152         marked = false;
153         numLinks = 0;
154         linkedNodes = (MapNode**) calloc(LIST_CHUNK, sizeof(MapNode*));
155     }
156
157     MapNode::MapNode(int x, int y, int z, int numNeighbours, MapNode** neighbours
            ) : PtNode (x,y,z)
158     {
159         predecessor = NULL;
160         hazardA = NULL;
161         hazardB = NULL;
162         marked = false;
163         numLinks = numNeighbours;
164         linkedNodes = neighbours;
165     }
```

```
166
167     MapNode::MapNode(Point xyz, int numNeighbours, MapNode** neighbours) :
            PtNode (xyz)
168     {
169         predecessor = NULL;
170         hazardA = NULL;
171         hazardB = NULL;
172         marked = false;
173         numLinks = numNeighbours;
174         linkedNodes = neighbours;
175     }
176
177     MapNode::~MapNode()
178     {
179         predecessor = NULL;
180         hazardA = NULL;
181         hazardB = NULL;
182         marked = false;
183
184         for(int i = 0; i < numLinks; i++)
185             linkedNodes[i]->BreakLink(this);
186
187         numLinks = -1;
188         free(linkedNodes);
189         linkedNodes = NULL;
190     }
191
192     /** Returns num of links the node has to others
193      */
194     int MapNode::GetNumLinks()
195     {
196         return numLinks;
197     }
198
199     /** Adds link to another node, if not already linked, and increments numLinks
200      */
201     void MapNode::AddLink(MapNode* node)
202     {
203         bool found = false;
204
205         /** find node, shift all other up */
206         for(int i = 0; i < numLinks; i++)
207         {
208             if(linkedNodes[i] == node)
209                 found = true;
210         }
211
```

```
212         if(!found)
213         {
214             // used below rather than if(numLinks%LIST_CHUNK == 0) cos that would
                    trip on numLinks = 0
215             if((numLinks-1)%LIST_CHUNK == LIST_CHUNK -1)
216             {
217                 void* tmp = realloc(linkedNodes, (numLinks + LIST_CHUNK) * sizeof
                        (MapNode*));
218                 if(tmp != NULL)
219                     linkedNodes = (MapNode**) tmp;
220                 else
221                     if(SHOW_NODE_ERRORS) printf("Error - Could not reallocate
                            memory for linkedNode\n");
222             }
223             linkedNodes[numLinks++] = node;
224         }
225     }
226
227     /** Breaks link to another node and decrements numLinks.
228      */
229     void MapNode::BreakLink(MapNode* node)
230     {
231         bool alreadyFound = false;
232
233         /** find node, shift all other up */
234         for(int i = 0; i < numLinks; i++)
235         {
236             if(alreadyFound && i + 1 < numLinks)
237                 linkedNodes[i] = linkedNodes[i+1];
238             else if(linkedNodes[i] == node)
239             {
240                 alreadyFound = true;
241                 linkedNodes[i] = linkedNodes[i+1];
242             }
243         }
244
245         if(alreadyFound)
246         {
247             linkedNodes[--numLinks] = NULL;
248             if((numLinks-1)%LIST_CHUNK == LIST_CHUNK -1)
249             {
250                 void* tmp = realloc(linkedNodes, numLinks * sizeof(MapNode*));
251                 if(tmp != NULL)
252                     linkedNodes = (MapNode**) tmp;
253                 else
254                     if(SHOW_NODE_ERRORS) printf("Error - Could not reallocate
                            memory for linkedNodes\n");
```

```
255             }
256         }
257         else
258         {
259             if(SHOW_NODE_WARNINGS) printf("Warning - Could not find node in list,
                    such that it could be unlinked\n");
260         }
261     }
262
263     /** Returns one of the nodes linked to
264      */
265     MapNode* MapNode::FollowLink(int num)
266     {
267         if(num >= numLinks || num < 0)
268         {
269             return NULL;//error
270         }
271         else
272             return linkedNodes[num];
273     }
274
275     Face_handle MapNode::GetHazardTriA()
276     {
277         if(typeid(hazardA) == typeid(Face_handle))
278             return hazardA;
279         else
280             return NULL;
281     }
282
283     Face_handle MapNode::GetHazardTriB()
284     {
285         return hazardB;
286     }
287
288
289     /** Check whether node has been marked during path planning
290      */
291     bool MapNode::CheckForMark()
292     {
293         return marked;
294     }
295
296     /** Mark node to indicate it has already been visited in path planning
297      */
298     void MapNode::MarkNode()
299     {
300         marked = true;
```

```
301     }
302
303     /********************************************** GridNode Object
            ***********************************************/
304
305     GridNode::GridNode() : PtNode()
306     {
307         predecessor = NULL;
308         marked = false;
309         hType = NONE;
310         clearance = true;
311         numLinks = 0;
312         linkedNodes = (GridNode**) calloc(LIST_CHUNK, sizeof(GridNode*));
313     }
314
315     GridNode::GridNode(int x, int y, int z) : PtNode(x,y,z)
316     {
317         predecessor = NULL;
318         marked = false;
319         hType = NONE;
320         clearance = true;
321         numLinks = 0;
322         linkedNodes = (GridNode**) calloc(LIST_CHUNK, sizeof(GridNode*));
323     }
324
325     GridNode::GridNode(Point xyz) : PtNode(xyz)
326     {
327         predecessor = NULL;
328         marked = false;
329         hType = NONE;
330         clearance = true;
331         numLinks = 0;
332         linkedNodes = (GridNode**) calloc(LIST_CHUNK, sizeof(GridNode*));
333     }
334
335     GridNode::GridNode(int x, int y, int z, HAZARDS hazardType, bool clear) :
            PtNode(x,y,z)
336     {
337         predecessor = NULL;
338         marked = false;
339         hType = hazardType;
340         clearance = clear;
341         numLinks = 0;
342         linkedNodes = (GridNode**) calloc(LIST_CHUNK, sizeof(GridNode*));
343     }
344
345     GridNode::GridNode(Point xyz, HAZARDS hazardType, bool clear) : PtNode(xyz)
```

```
346     {
347         predecessor = NULL;
348         marked = false;
349         hType = hazardType;
350         clearance = clear;
351         numLinks = 0;
352         linkedNodes = (GridNode**) calloc(LIST_CHUNK, sizeof(GridNode*));
353     }
354
355     GridNode::GridNode(int x, int y, int z, int numNeighbours, GridNode**
            neighbours) : PtNode (x,y,z)
356     {
357         predecessor = NULL;
358         marked = false;
359         hType = NONE;
360         clearance = true;
361         numLinks = numNeighbours;
362         linkedNodes = neighbours;
363     }
364
365     GridNode::GridNode(Point xyz, int numNeighbours, GridNode** neighbours) :
            PtNode (xyz)
366     {
367         predecessor = NULL;
368         marked = false;
369         hType = NONE;
370         clearance = true;
371         numLinks = numNeighbours;
372         linkedNodes = neighbours;
373     }
374
375
376     GridNode::~GridNode()
377     {
378         predecessor = NULL;
379         hType = NONE;
380         marked = false;
381         clearance = true;
382
383         for(int i = 0; i < numLinks; i++)
384             linkedNodes[i]->BreakLink(this);
385
386         numLinks = -1;
387         free(linkedNodes);
388         linkedNodes = NULL;
389     }
390
```

```
391     /** Returns num of links the node has to others
392      */
393     int GridNode::GetNumLinks()
394     {
395         return numLinks;
396     }
397 
398     /** Adds link to another node, if not already linked, and increments numLinks
399      */
400     void GridNode::AddLink(GridNode* node)
401     {
402         bool found = false;
403 
404         /** find node, shift all other up */
405         for(int i = 0; i < numLinks; i++)
406         {
407             if(linkedNodes[i] == node)
408                 found = true;
409         }
410 
411         if(!found)
412         {
413             // used below rather than if(numLinks%LIST_CHUNK == 0) cos that would
                    trip on numLinks = 0
414             if((numLinks-1)%LIST_CHUNK == LIST_CHUNK -1)
415             {
416                 void* tmp = realloc(linkedNodes, (numLinks + LIST_CHUNK) * sizeof
                        (GridNode*));
417                 if(tmp != NULL)
418                     linkedNodes = (GridNode**) tmp;
419                 else
420                     if(SHOW_NODE_ERRORS) printf("Error - Could not reallocate
                            memory for linkedNode\n");
421             }
422             linkedNodes[numLinks++] = node;
423         }
424     }
425 
426     /** Breaks link to another node and decrements numLinks.
427      */
428     void GridNode::BreakLink(GridNode* node)
429     {
430         bool alreadyFound = false;
431 
432         /** find node, shift all other up */
433         for(int i = 0; i < numLinks; i++)
434         {
```

```
435             if(alreadyFound && i + 1 < numLinks)
436                 linkedNodes[i] = linkedNodes[i+1];
437             else if(linkedNodes[i] == node)
438             {
439                 alreadyFound = true;
440                 linkedNodes[i] = linkedNodes[i+1];
441             }
442         }
443 
444         if(alreadyFound)
445         {
446             linkedNodes[--numLinks] = NULL;
447             if((numLinks-1)%LIST_CHUNK == LIST_CHUNK -1)
448             {
449                 void* tmp = realloc(linkedNodes, numLinks * sizeof(GridNode*));
450                 if(tmp != NULL)
451                     linkedNodes = (GridNode**) tmp;
452                 else
453                     if(SHOW_NODE_ERRORS) printf("Error - Could not reallocate
                        memory for linkedNodes\n");
454             }
455         }
456         else
457         {
458             if(SHOW_NODE_WARNINGS) printf("Warning - Could not find node in list,
                ,such that it could be unlinked\n");
459         }
460     }
461 
462     void GridNode::ExpandLinks(int direction, int stepSize, GridNode prevNode)
463     {
464         if(direction == 0 || direction == 1)
465         {
466             for(int i = 0; i < prevNode.numLinks; i++)
467             {
468                 if(*prevNode.FollowLink(i) == prevNode.GetPoint() + Vector(-
                    stepSize, 0, stepSize))
469                 {
470                     AddLink(prevNode.FollowLink(i));
471                     prevNode.FollowLink(i)->AddLink(this);
472                     break;
473                 }
474             }
475         }
476 
477         if(direction == 1 || direction == 2)
478         {
```

```
479             for(int i = 0; i < prevNode.numLinks; i++)
480             {
481                 if(*prevNode.FollowLink(i) == prevNode.GetPoint() + Vector(
                        stepSize, 0, stepSize))
482                 {
483                     AddLink(prevNode.FollowLink(i));
484                     prevNode.FollowLink(i)->AddLink(this);
485                     break;
486                 }
487             }
488         }
489 
490         if(direction == 2 || direction == 3)
491         {
492             for(int i = 0; i < prevNode.numLinks; i++)
493             {
494                 if(*prevNode.FollowLink(i) == prevNode.GetPoint() + Vector(
                        stepSize, 0, -stepSize))
495                 {
496                     AddLink(prevNode.FollowLink(i));
497                     prevNode.FollowLink(i)->AddLink(this);
498                     break;
499                 }
500             }
501         }
502 
503         if(direction == 0 || direction == 3)
504         {
505             for(int i = 0; i < prevNode.numLinks; i++)
506             {
507                 if(*prevNode.FollowLink(i) == prevNode.GetPoint() + Vector(-
                        stepSize, 0, -stepSize))
508                 {
509                     AddLink(prevNode.FollowLink(i));
510                     prevNode.FollowLink(i)->AddLink(this);
511                     break;
512                 }
513             }
514         }
515     }
516 
517     /** Returns one of the nodes linked to
518      */
519     GridNode* GridNode::FollowLink(int num)
520     {
521         if(num >= numLinks || num < 0)
522         {
```

```
523             return NULL; //error
524         }
525         else
526             return linkedNodes[num];
527     }
528
529     /** Check whether node has been marked during path planning
530      */
531     bool GridNode::CheckForMark()
532     {
533         return marked;
534     }
535
536     /** Check whether node is considered to have enough clearance around it to be
            traversable
537      */
538     bool GridNode::CheckForClearance()
539     {
540         return clearance;
541     }
542
543     /** Mark node to indicate it has already been visited in path planning
544      */
545     void GridNode::MarkNode()
546     {
547         marked = true;
548     }
549
550     /************************************************ ASTAR_MapNode Object
            ************************************************/
551
552     ASTAR_MapNode::ASTAR_MapNode() : MapNode()
553     {
554         open = false, closed = false;
555         predecessor = NULL;
556         successor = NULL;
557         f_val = -1, g_val = -1, h_val = -1;
558         linkedNodes = (ASTAR_MapNode**) calloc(LIST_CHUNK, sizeof(ASTAR_MapNode*)
                );
559     }
560
561     ASTAR_MapNode::ASTAR_MapNode(int x, int y, int z) : MapNode(x,y,z)
562     {
563         open = false, closed = false;
564         predecessor = NULL;
565         successor = NULL;
566         f_val = -1, g_val = -1, h_val = -1;
```

```
567         linkedNodes = (ASTAR_MapNode**) calloc(LIST_CHUNK, sizeof(ASTAR_MapNode*)
                );
568     }
569
570     ASTAR_MapNode::ASTAR_MapNode(Point xyz) : MapNode(xyz)
571     {
572         open = false, closed = false;
573         predecessor = NULL;
574         successor = NULL;
575         f_val = -1, g_val = -1, h_val = -1;
576         linkedNodes = (ASTAR_MapNode**) calloc(LIST_CHUNK, sizeof(ASTAR_MapNode*)
                );
577     }
578
579     ASTAR_MapNode::ASTAR_MapNode(int x, int y, int z, Face_handle nodeA,
            Face_handle nodeB) : MapNode(x,y,z, nodeA, nodeB)
580     {
581         open = false, closed = false;
582         predecessor = NULL;
583         successor = NULL;
584         f_val = -1, g_val = -1, h_val = -1;
585         linkedNodes = (ASTAR_MapNode**) calloc(LIST_CHUNK, sizeof(ASTAR_MapNode*)
                );
586     }
587
588     ASTAR_MapNode::ASTAR_MapNode(Point xyz, Face_handle nodeA, Face_handle nodeB)
            : MapNode(xyz, nodeA, nodeB)
589     {
590         open = false, closed = false;
591         predecessor = NULL;
592         successor = NULL;
593         f_val = -1, g_val = -1, h_val = -1;
594         linkedNodes = (ASTAR_MapNode**) calloc(LIST_CHUNK, sizeof(ASTAR_MapNode*)
                );
595     }
596
597     ASTAR_MapNode::~ASTAR_MapNode()
598     {
599         open = false, closed = false;
600         predecessor = NULL;
601         successor = NULL;
602         f_val = -1, g_val = -1, h_val = -1;
603
604         for(int i = 0; i < numLinks; i++)
605         {
606             linkedNodes[i]->BreakLink(this);
607         }
```

```
608
609         numLinks = -1;
610         free(linkedNodes);
611         linkedNodes = NULL;
612     }
613
614     /** Adds link to another node, if not already linked, and increments numLinks
615      */
616     void ASTAR_MapNode::AddLink(ASTAR_MapNode* node)
617     {
618         bool found = false;
619
620         /** find node, shift all other up */
621         for(int i = 0; i < numLinks; i++)
622         {
623             if(linkedNodes[i] == node)
624                 found = true;
625         }
626
627         if(!found)
628         {
629             if((numLinks-1)%LIST_CHUNK == LIST_CHUNK -1)
630             {
631                 void* tmp = realloc(linkedNodes, (numLinks + LIST_CHUNK) * sizeof
                        (ASTAR_MapNode*));
632                 if(tmp != NULL)
633                     linkedNodes = (ASTAR_MapNode**) tmp;
634                 else
635                     printf("Error - Could not reallocate memory for linkedNode\n"
                            );
636             }
637             linkedNodes[numLinks++] = node;
638         }
639
640         if(numLinks > 100)
641             int cat = 7;
642
643     }
644
645     /** Breaks link to another node and decrements numLinks.
646      */
647     void ASTAR_MapNode::BreakLink(ASTAR_MapNode* node)
648     {
649         bool alreadyFound = false;
650
651         /** find node, shift all other up */
652         for(int i = 0; i < numLinks; i++)
```

```
653         {
654             if(alreadyFound && i + 1 < numLinks)
655                 linkedNodes[i] = linkedNodes[i+1];
656             else if(linkedNodes[i] == node)
657             {
658                 alreadyFound = true;
659                 if(i + 1 < numLinks)
660                     linkedNodes[i] = linkedNodes[i+1];
661             }
662         }
663         if(alreadyFound)
664         {
665             linkedNodes[--numLinks] = NULL;
666             if((numLinks-1)%LIST_CHUNK == LIST_CHUNK -1)
667             {
668                 void* tmp = realloc(linkedNodes, numLinks * sizeof(ASTAR_MapNode
                        *));
669                 if(tmp != NULL)
670                     linkedNodes = (ASTAR_MapNode**) tmp;
671                 else
672                     if(SHOW_NODE_ERRORS) printf("Error - Could not reallocate
                            memory for linkedNodes\n");
673             }
674
675             if(predecessor == node)
676             {
677                 predecessor = NULL;
678                 if(open)
679                     int cat = 7;
680             }
681             if(successor == node)
682             {
683                 successor = NULL;
684                 if(closed)
685                     int cat = 7;
686             }
687         }
688         else
689         {
690             if(SHOW_NODE_WARNINGS) printf("Warning - Could not find node in list,
                    such that it could be unlinked\n");
691         }
692     }
693
694     /** Returns one of the nodes linked to
695      */
696     ASTAR_MapNode* ASTAR_MapNode::FollowLink(int num)
```

```
697     {
698         if(num >= numLinks || num < 0)
699         {
700             return NULL;//error
701         }
702         else
703             return linkedNodes[num];
704     }
705
706     /** Returns one of the nodes linked to
707      */
708     void ASTAR_MapNode::ResetOpenClosed()
709     {
710         open = false;
711         closed = false;
712     }
713
714     /********************************************** ASTAR_GridNode Object
            **********************************************/
715
716     ASTAR_GridNode::ASTAR_GridNode() : GridNode()
717     {
718         open = false, closed = false;
719         predecessor = NULL;
720         successor = NULL;
721         f_val = -1, g_val = -1, h_val = -1;
722         linkedNodes = (ASTAR_GridNode**) calloc(LIST_CHUNK, sizeof(ASTAR_GridNode
                *));
723     }
724
725     ASTAR_GridNode::ASTAR_GridNode(int x, int y, int z) : GridNode(x,y,z)
726     {
727         open = false, closed = false;
728         predecessor = NULL;
729         successor = NULL;
730         f_val = -1, g_val = -1, h_val = -1;
731         linkedNodes = (ASTAR_GridNode**) calloc(LIST_CHUNK, sizeof(ASTAR_GridNode
                *));
732     }
733
734     ASTAR_GridNode::ASTAR_GridNode(Point xyz) : GridNode(xyz)
735     {
736         open = false, closed = false;
737         predecessor = NULL;
738         successor = NULL;
739         f_val = -1, g_val = -1, h_val = -1;
```

```
740         linkedNodes = (ASTAR_GridNode**) calloc(LIST_CHUNK, sizeof(ASTAR_GridNode
                *));
741     }
742
743     ASTAR_GridNode::ASTAR_GridNode(int x, int y, int z, HAZARDS hazardType, bool
            clearance) : GridNode(x,y,z, hazardType, clearance)
744     {
745         open = false, closed = false;
746         predecessor = NULL;
747         successor = NULL;
748         f_val = -1, g_val = -1, h_val = -1;
749         linkedNodes = (ASTAR_GridNode**) calloc(LIST_CHUNK, sizeof(ASTAR_GridNode
                *));
750     }
751
752     ASTAR_GridNode::ASTAR_GridNode(Point xyz, HAZARDS hazardType, bool clearance)
             : GridNode(xyz, hazardType, clearance)
753     {
754         open = false, closed = false;
755         predecessor = NULL;
756         successor = NULL;
757         f_val = -1, g_val = -1, h_val = -1;
758         linkedNodes = (ASTAR_GridNode**) calloc(LIST_CHUNK, sizeof(ASTAR_GridNode
                *));
759     }
760
761     ASTAR_GridNode::~ASTAR_GridNode()
762     {
763         open = false, closed = false;
764         predecessor = NULL;
765         successor = NULL;
766         f_val = -1, g_val = -1, h_val = -1;
767
768         for(int i = 0; i < numLinks; i++)
769         {
770             linkedNodes[i]->BreakLink(this);
771         }
772
773         numLinks = -1;
774         free(linkedNodes);
775         linkedNodes = NULL;
776     }
777
778     /** Adds link to another node, if not already linked, and increments numLinks
779      */
780     void ASTAR_GridNode::AddLink(ASTAR_GridNode* node)
781     {
```

```
782         bool found = false;
783 
784         /** find node, shift all other up */
785         for(int i = 0; i < numLinks; i++)
786         {
787             if(linkedNodes[i] == node)
788                 found = true;
789         }
790 
791         if(!found)
792         {
793             if((numLinks-1)%LIST_CHUNK == LIST_CHUNK -1)
794             {
795                 void* tmp = realloc(linkedNodes, (numLinks + LIST_CHUNK) * sizeof
                        (ASTAR_GridNode*));
796                 if(tmp != NULL)
797                     linkedNodes = (ASTAR_GridNode**) tmp;
798                 else
799                     printf("Error - Could not reallocate memory for linkedNode\n"
                            );
800             }
801             linkedNodes[numLinks++] = node;
802         }
803 
804         if(numLinks > 100)
805             int cat = 7;
806 
807     }
808 
809     /** Breaks link to another node and decrements numLinks.
810      */
811     void ASTAR_GridNode::BreakLink(ASTAR_GridNode* node)
812     {
813         bool alreadyFound = false;
814 
815         /** find node, shift all other up */
816         for(int i = 0; i < numLinks; i++)
817         {
818             if(alreadyFound && i + 1 < numLinks)
819                 linkedNodes[i] = linkedNodes[i+1];
820             else if(linkedNodes[i] == node)
821             {
822                 alreadyFound = true;
823                 if(i + 1 < numLinks)
824                     linkedNodes[i] = linkedNodes[i+1];
825             }
826         }
```

```
827         if(alreadyFound)
828         {
829             linkedNodes[--numLinks] = NULL;
830             if((numLinks-1)%LIST_CHUNK == LIST_CHUNK -1)
831             {
832                 void* tmp = realloc(linkedNodes, numLinks * sizeof(ASTAR_GridNode
                        *));
833                 if(tmp != NULL)
834                     linkedNodes = (ASTAR_GridNode**) tmp;
835                 else
836                     if(SHOW_NODE_ERRORS) printf("Error - Could not reallocate
                            memory for linkedNodes\n");
837             }
838 
839             if(predecessor == node)
840             {
841                 predecessor = NULL;
842                 if(open)
843                     int cat = 7;
844             }
845             if(successor == node)
846             {
847                 successor = NULL;
848                 if(closed)
849                     int cat = 7;
850             }
851         }
852         else
853         {
854             if(SHOW_NODE_WARNINGS) printf("Warning - Could not find node in list,
                    such that it could be unlinked\n");
855         }
856     }
857 
858     void ASTAR_GridNode::ExpandLinks(int direction, int stepSize, ASTAR_GridNode
            prevNode)
859     {
860         if(direction == 0 || direction == 1)
861         {
862             for(int i = 0; i < prevNode.numLinks; i++)
863             {
864                 if(*prevNode.FollowLink(i) == prevNode.GetPoint() + Vector(-
                        stepSize, 0, stepSize))
865                 {
866                     AddLink(prevNode.FollowLink(i));
867                     prevNode.FollowLink(i)->AddLink(this);
868                     break;
```

```
869                 }
870             }
871         }
872
873         if(direction == 1 || direction == 2)
874         {
875             for(int i = 0; i < prevNode.numLinks; i++)
876             {
877                 if(*prevNode.FollowLink(i) == prevNode.GetPoint() + Vector(
                        stepSize, 0, stepSize))
878                 {
879                     AddLink(prevNode.FollowLink(i));
880                     prevNode.FollowLink(i)->AddLink(this);
881                     break;
882                 }
883             }
884         }
885
886         if(direction == 2 || direction == 3)
887         {
888             for(int i = 0; i < prevNode.numLinks; i++)
889             {
890                 if(*prevNode.FollowLink(i) == prevNode.GetPoint() + Vector(
                        stepSize, 0, -stepSize))
891                 {
892                     AddLink(prevNode.FollowLink(i));
893                     prevNode.FollowLink(i)->AddLink(this);
894                     break;
895                 }
896             }
897         }
898
899         if(direction == 0 || direction == 3)
900         {
901             for(int i = 0; i < prevNode.numLinks; i++)
902             {
903                 if(*prevNode.FollowLink(i) == prevNode.GetPoint() + Vector(-
                        stepSize, 0, -stepSize))
904                 {
905                     AddLink(prevNode.FollowLink(i));
906                     prevNode.FollowLink(i)->AddLink(this);
907                     break;
908                 }
909             }
910         }
911     }
912
```

```
913     /** Returns one of the nodes linked to
914      */
915     ASTAR_GridNode* ASTAR_GridNode::FollowLink(int num)
916     {
917         if(num >= numLinks || num < 0)
918         {
919             return NULL;//error
920         }
921         else
922             return linkedNodes[num];
923     }
924 
925     /** Returns one of the nodes linked to
926      */
927     void ASTAR_GridNode::ResetOpenClosed()
928     {
929         open = false;
930         closed = false;
931     }
```

A.6 Robot

A.6.1 Robot.h

```
1  #ifndef ROBOT_H_
2  #define ROBOT_H_
3  
4  const float TANOFANGLE = 0.577; // equal to tan(SLOPEANGLE) which is angle too
       steep to ascend
5  // const float TANOFINVANGLE = 1.732;
6  const int SLOPEANGLE = 30;
7  const int WALLANGLE = 60;
8  // const float INVSHEERFACE = 0.577; // 0.577 = tan(90 - WALLANGLE) where theta
       is gradient great than which are considered sheerface, in this case 60degrees
9  
10 const float VISIBLE_DIST_MM = 10000.0f;
11 const float CLEARANCEBUFFER_MM = 100.0f;
12 const float MOVE_INCREMENTS_MM = 100.0f;//equates to move distance
13 const int STEPS_BETWEEN_EVAL = 1;
14 const int ROBOTLENGTH_MM = 500;
15 const int ROBOTWIDTH_MM = 300;
16 const int AXLESEP_MM = 300; //distance between axles of robot
17 const int WHEELWIDTH_MM = 50;
18 const int WHEELRADIUS_MM = 100;
```

```
19 const int CAMERA_HEIGHT = WHEELRADIUS_MM; // if designed to run either way up.
       possibly higher if not designed to handle flipping. scan would have to be
       adjusted for that
20
21 const bool KNOWN = false;
22 const bool KNOWN_NODES = false;
23
24 class Robot;
25
26 #include <time.h>
27 #include "Nodes.h"
28 #include "World.h"
29 #include "SimulationEnvironment.h"
30
31 enum ALG {NO_ALG, HUGGER, STATE, EIGHT, DFS, BFS, GREEDY, ASTAR, LPASTAR, DSTAR};
32
33 const ALG algorithm = LPASTAR;
34 // #define ALG_NODE_TYPE MapNode
35 // #define ALG_NODE_TYPE GridNode
36 #define ALG_NODE_TYPE ASTAR_MapNode
37 // #define ALG_NODE_TYPE ASTAR_GridNode
38
39 class Robot{
40 public:
41     virtual ~Robot();
42     /**
43      * gpsCoords are coords the world should start relative to
44      */
45     Robot(Coord gpsCoords, Coord location, Coord destination);
46     Robot(Coord gpsCoords, Coord location, Coord destination,
          SimulationEnvironment* simEnviron);
47
48     void GatherData();
49     void GatherData(Coord* scannedData, int numScanPoints);
50     void GatherData(SimulationEnvironment* simEnv, int inputQualityLevel);
51
52     template <class NODE>   void GetMapNodes(NODE** currLoc, NODE** goal);
53 //  template <class NODE>   void GetGridNodes(NODE** currLoc, NODE** goal);
54
55     void PlanPath();
56     template <class TRI, class PT>  void EdgeHugger();
57     Coord CheckEdgePath(Coord start, Coord* testPos, int numPoints, Coord*
          pastTestPoints);
58
59     bool Halt();
60     bool NaviDone();
61     Coord MoveDirection();
```

```
62      void SetDirection(Coord direction);
63      Coord GetDirection();
64      Coord GetPosition();
65      Point GetNext();
66      Point GetGoalPt();
67      Point GetStartPt();
68      Coord GetGoal();
69      Coord GetLastMove();
70      void AdjustPosition(Coord change);
71      void AdjustPosition(int xChange, int yChange, int zChange);
72
73      void SetPath(Point* path);
74      void SetNumSteps(int steps);
75
76      void SaveInfo(int** data, Coord gpsReference);
77
78      void DisplayPlannedPath();
79
80      void RecordData(char* filename, int x, int y, int z);
81      void RecordMoreData(char* filename, int x, int y, int z);
82      void RecordEndOfData(char* filename, int x, int y, int z);
83
84      World<ALG_NODE_TYPE>* terrain;
85      int numMoves;
86      int numBadPath;
87      int consecutiveTimesStationary;
88
89  private:
90      void AddNodeToPath(MapNode* node);
91      void RemoveNodeFromPath();
92
93      void ShowContents(int ** dataPoints);
94
95      int consecutiveTimesAlternating;
96
97      bool gridGraph;
98      bool reachable;
99      Coord currentPos;
100     Coord directionFacing;
101     Coord lastMove;
102     Coord gpsRefCoords;
103     Coord goal;
104
105     Point startPt;
106     Point currentPt;
107     Point goalPt;
108
```

```
109     int algorithmUsed;
110
111     Point* plannedPath;
112     int numStepsInPath;
113
114     ///World* terrain;
115     SimulationEnvironment* sim;
116 };
117
118     template <class NODE>
119     void Robot::GetMapNodes(NODE** currLoc, NODE** goal)
120     {
121         //take ptrs, set to values rather than given allocated array
122         int numNodes = 0;
123         Point dirVect = ConvertPointType<Coord, Point>(GetDirection());
124
125         if(KNOWN_NODES)
126             terrain->GenPreMadeMapNodes(dirVect);
127         else
128         {
129     //      ///if not grid based
130     //      if(gType == MAP)
131     //      {
132                 HazardNode *hNodes = (HazardNode*) calloc(ARRAY_CHUNK, sizeof(
                        HazardNode));
133                 terrain->GenHazardNodes(&hNodes, &numNodes, dirVect);
134                 terrain->GenMapNodes(hNodes, numNodes, dirVect);
135                 free(hNodes);
136                 hNodes = NULL;
137     //      }
138     //      ///if is grid based
139     //      else if(gType == GRID)
140     //      {
141     //          terrain->GenGridNodes(dirVect);
142     //      }
143         }
144
145         *currLoc = (NODE*) terrain->GetCurrentNode();
146         *goal =  (NODE*) terrain->GetGoalNode();
147     }
148
149     /** Simple planning algorithm which follows the boundaries of hazards
150         Have it check ahead of self, to allow for size of robot
151         Have it save end waypoint, though just checking ok to move forward 10 in
                that direction
152     */
153     template <class TRI, class PT>
```

```
154     void Robot::EdgeHugger()
155     {
156         int height = goal.zVal;
157 
158         PT tGoal = ConvertPointType<Coord, PT>(goal);
159         PT tCurrentPos = ConvertPointType<Coord, PT>(currentPos);
160         PT directionVector = tGoal - (tCurrentPos - PT(0,0,0));
161         float magn = CalculateDirectDistance(currentPos, goal, true);
162         directionVector = NormaliseVector<PT>(directionVector ,
                MOVE_INCREMENTS_MM, true);
163         PT normVector = CrossProduct(directionVector, directionVector + (PT
                (0,0,10) - PT(0,0,0)));
164         normVector = NormaliseVector<PT>(normVector , ROBOTWIDTH_MM/2, false);
165 
166         PT tNewPos = tCurrentPos + (directionVector - PT(0,0,0));
167         Coord newPos = ConvertPointType<PT, Coord>(tNewPos);
168         TRI newTri;
169 
170         numStepsInPath = 1;
171         bool pathClear = terrain->CheckBetweenPoints(tCurrentPos, tNewPos + (
                directionVector - PT(0,0,0)), true, normVector);
172 
173         if(terrain->GetTri(&newTri, &tNewPos))
174         {
175             if(pathClear && newTri->CheckValidTri() && !newTri->hazardType)
176                 plannedPath[0] = ConvertPointType<PT, Point>(tNewPos);
177             else
178             {
179                 Coord* testPts = (Coord*) calloc(ARRAY_CHUNK, sizeof(Coord));
180                 newPos = CheckEdgePath(currentPos, &(ConvertPointType<PT, Coord>(
                        tNewPos + (directionVector - PT(0,0,0)))), 0, testPts);
181                 plannedPath[0] = ConvertPointType<Coord, Point>(newPos);
182                 free(testPts);
183                 testPts = NULL;
184             }
185         }
186         else
187         {
188             TRI currentTri = TRI();
189             if(pathClear)
190                 plannedPath[0] = ConvertPointType<PT, Point>(tNewPos);
191             else if(terrain->GetTri(&currentTri, &PT(currentPos.x(), currentPos.y
                    (), currentPos.z())) && currentTri->CheckValidTri())
192             {
193                 int dist1 = 999, dist2 = 999, dist3 = 999;
194 
```

```
195                 ///This needs to be improved/changes. Doesn't necessarily pick
                        right neighbour to go through/towards
196                 if(currentTri->neighbor(0) != NULL)
197                 {
198                     dist1 =  CalculateDirectDistance(newPos, CalcTriCentre(
                            ConvertPointType<PT, Coord>(*currentTri->neighbor(0)->
                            GetPoint(0)), ConvertPointType<PT, Coord>(*currentTri->
                            neighbor(0)->GetPoint(1)), ConvertPointType<PT, Coord>(*
                            currentTri->neighbor(0)->GetPoint(2))), false);
199                 }
200                 if(currentTri->neighbor(1) != NULL)
201                     dist2 = CalculateDirectDistance(newPos, CalcTriCentre(
                            ConvertPointType<PT, Coord>(*currentTri->neighbor(1)->
                            GetPoint(0)), ConvertPointType<PT, Coord>(*currentTri->
                            neighbor(1)->GetPoint(1)), ConvertPointType<PT, Coord>(*
                            currentTri->neighbor(1)->GetPoint(2))), false);
202                 if(currentTri->neighbor(2) != NULL)
203                     dist3 = CalculateDirectDistance(newPos, CalcTriCentre(
                            ConvertPointType<PT, Coord>(*currentTri->neighbor(2)->
                            GetPoint(0)), ConvertPointType<PT, Coord>(*currentTri->
                            neighbor(2)->GetPoint(1)), ConvertPointType<PT, Coord>(*
                            currentTri->neighbor(2)->GetPoint(2))), false);
204
205                 if(dist1 == 999 && dist2 == 999 && dist3 == 999)
206                     plannedPath[0] = ConvertPointType<PT, Point>(tNewPos);
207
208                 Coord* testPts = (Coord*) calloc(30, sizeof(Coord));
209
210                 //would have issues if a neighbour exists which has centre over
                        999 away from current, though should not occur
211                 if(dist1 <= dist2 && dist1 <= dist3 && currentTri->neighbor(0)->
                        hazardType)
212                 {
213                         plannedPath[0] = ConvertPointType<Coord, Point>(
                                CheckEdgePath(currentPos, &CalcTriCentre(
                                ConvertPointType<PT, Coord>(*currentTri->neighbor(0)
                                ->GetPoint(0)), ConvertPointType<PT, Coord>(*
                                currentTri->neighbor(0)->GetPoint(1)),
                                ConvertPointType<PT, Coord>(*currentTri->neighbor(0)
                                ->GetPoint(2))), 0, testPts));
214                         pathClear = true;
215                 }
216                 if(!pathClear && dist2 <= dist1 && dist2 <= dist3 && currentTri->
                        neighbor(1)->hazardType)
217                 {
218                         plannedPath[0] = ConvertPointType<Coord, Point>(
                                CheckEdgePath(currentPos, &CalcTriCentre(
```

```
                                ConvertPointType<PT, Coord>(*currentTri->neighbor(1)
                                ->GetPoint(0)), ConvertPointType<PT, Coord>(*
                                currentTri->neighbor(1)->GetPoint(1)),
                                ConvertPointType<PT, Coord>(*currentTri->neighbor(1)
                                ->GetPoint(2))), 0, testPts));
219                         pathClear = true;
220                 }
221                 if(!pathClear && dist3 < dist1 && dist3 < dist2 && currentTri->
                        neighbor(2)->hazardType)
222                 {
223                         plannedPath[0] = ConvertPointType<Coord, Point>(
                                CheckEdgePath(currentPos, &CalcTriCentre(
                                ConvertPointType<PT, Coord>(*currentTri->neighbor(2)
                                ->GetPoint(0)), ConvertPointType<PT, Coord>(*
                                currentTri->neighbor(2)->GetPoint(1)),
                                ConvertPointType<PT, Coord>(*currentTri->neighbor(2)
                                ->GetPoint(2))), 0, testPts));
224                         pathClear = true;
225                 }
226
227                 free(testPts);
228                 testPts = NULL;
229
230                 if(!pathClear)
231                     plannedPath[0] = currentPt;
232             }
233             else
234                 plannedPath[0] = ConvertPointType<PT, Point>(tNewPos);
235         }
236     }
237
238
239 #endif /*ROBOT_H_*/
```

A.6.2 Robot.cpp

```
1  /*************************************************************************
2   *   Copyright (C) 2007 by Michael Douglas Hasler   *
3   *   redheadpunk@gmail.com   *
4   *                                                                       *
5   *   This program is free software; you can redistribute it and/or modify  *
6   *   it under the terms of the GNU General Public License as published by  *
7   *   the Free Software Foundation; either version 2 of the License, or     *
8   *   (at your option) any later version.                                   *
9   *                                                                       *
10  *   This program is distributed in the hope that it will be useful,       *
```

```
11  *   but WITHOUT ANY WARRANTY; without even the implied warranty of       *
12  *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the        *
13  *   GNU General Public License for more details.                         *
14  *                                                                        *
15  *   You should have received a copy of the GNU General Public License    *
16  *   along with this program; if not, write to the                        *
17  *   Free Software Foundation, Inc.,                                      *
18  *   59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.            *
19  **************************************************************************/
20 #include "adv_math.h"
21 #include "Algorithms.h"
22 #include "Robot.h"
23
24     /** Constructor for Robot object
25      * Sets local variables of Robot and creates a World object
26      */
27     Robot::Robot(Coord gpsCoords, Coord location, Coord destination)
28     {
29         directionFacing = destination - location;
30         directionFacing.zVal = 0;
31         currentPos = location;
32         goal = destination;
33         gpsRefCoords = gpsCoords;
34         startPt =  Point(location.x(), location.y(), location.z());
35         currentPt = startPt;
36         goalPt = Point(destination.x(), destination.y(), destination.z());
37         numMoves = 0;
38         numBadPath = 0;
39         consecutiveTimesStationary = 0;
40         consecutiveTimesAlternating = 0;
41         numStepsInPath = 0;
42         plannedPath = (Point*) calloc(LIST_CHUNK, sizeof(Point));
43         reachable = true;
44         terrain = new World<ALG_NODE_TYPE>(gpsRefCoords, currentPos, goal);
45         terrain->SetHazardIdentParam(SLOPEANGLE,  WALLANGLE, ROBOTLENGTH_MM,
               ROBOTWIDTH_MM, AXLESEP_MM, WHEELRADIUS_MM, WHEELWIDTH_MM,
               CLEARANCEBUFFER_MM, MOVE_INCREMENTS_MM, VISIBLE_DIST_MM);
46         if (KNOWN)
47             terrain->LoadKnown(KNOWN_NODES);
48         sim = NULL;
49         algorithmUsed = NO_ALG;
50     }
51
52     /** Constructor for Robot object
53      * Sets local variables of Robot and creates a World object
54      */
```

```
55  Robot::Robot(Coord gpsCoords, Coord location, Coord destination,
        SimulationEnvironment* simEnviron)
56  {
57      directionFacing = destination - location;
58      directionFacing.zVal = 0;
59      currentPos = location;
60      goal = destination;
61      gpsRefCoords = gpsCoords;
62      startPt =  Point(location.x(), location.y(), location.z());
63      currentPt = startPt;
64      goalPt = Point(destination.x(), destination.y(), destination.z());
65      numMoves = 0;
66      numBadPath = 0;
67      consecutiveTimesStationary = 0;
68      consecutiveTimesAlternating = 0;
69      numStepsInPath = 0;
70      plannedPath = (Point*) calloc(LIST_CHUNK, sizeof(Point*));
71      reachable = true;
72      terrain = new World<ALG_NODE_TYPE>(gpsRefCoords, currentPos, goal);
73      terrain->SetHazardIdentParam(SLOPEANGLE,  WALLANGLE, ROBOTLENGTH_MM,
            ROBOTWIDTH_MM, AXLESEP_MM, WHEELRADIUS_MM, WHEELWIDTH_MM,
            CLEARANCEBUFFER_MM, MOVE_INCREMENTS_MM, VISIBLE_DIST_MM);
74      if(KNOWN)
75          terrain->LoadKnown(KNOWN_NODES);
76      sim = simEnviron;
77      algorithmUsed = NO_ALG;
78  }
79
80  Robot::~Robot()
81  {
82      free(plannedPath);
83      plannedPath = NULL;
84      delete terrain;
85  }
86
87  void Robot::DisplayPlannedPath()
88  {
89      WriteLineToFile("path.txt", "wt", (Point**) NULL, 0);
90      for(int i = 1; i < numStepsInPath; i++)
91      {
92          Point* edge[2] = {&plannedPath[i - 1], &plannedPath[i]};
93          WriteLineToFile("path.txt", "at", edge, 2);
94      }
95
96      WriteColourChangeToFile("path.txt");
97      Point* edge[2] = {&goalPt, &(goalPt + Vector(0,2,0))};
98      WriteLineToFile("path.txt", "at", edge, 2);
```

```
99          edge[0] = &currentPt; edge[1] = &(currentPt - Vector(0,2,0));
100         WriteLineToFile("path.txt", "at", edge, 2);
101         edge[0] = &startPt; edge[1] = &(startPt - Vector(0,2,0));
102         WriteLineToFile("path.txt", "at", edge, 2);
103
104         display("path.txt");
105     }
106
107     void Robot::RecordData(char* filename, int x, int y, int z)
108     {
109         FILE* output = fopen(filename, "at");
110         time_t rawtime;
111         time(&rawtime);
112         fprintf(output,"Results from %s:\n", asctime(localtime(&rawtime)));
113         fprintf(output,"Coordinates \t%d\t%d\t%d\n", x,y,z);
114         fclose(output);
115     }
116
117     void Robot::RecordMoreData(char* filename, int x, int y, int z)
118     {
119         FILE* output = fopen(filename, "at");
120         fprintf(output,"Coordinates \t%d\t%d\t%d\n", x,y,z);
121         fclose(output);
122     }
123
124     void Robot::RecordEndOfData(char* filename, int x, int y, int z)
125     {
126         FILE* output = fopen(filename, "at");
127         time_t rawtime;
128         time(&rawtime);
129         fprintf(output,"Coordinates \t%d\t%d\t%d\n\n", x,y,z);
130         fprintf(output,"Trial ending at %s:\n", asctime(localtime(&rawtime)));
131         fprintf(output,"\n\n\n");
132         fclose(output);
133     }
134
135     /** One of the two main functions of robot.
136      * This calls the functions to get and process/prepare data
137      */
138     void Robot::GatherData()
139     {
140         int size = 121;
141         Coord* scannedData = (Coord*) calloc(size, sizeof(Coord));
142         int numScanPoints = 0;
143         if(sim != NULL)
144             sim->ScanViewPoints(&scannedData, &numScanPoints, currentPos,
                    directionFacing);
```

```
145         else
146             printf("Error - Sim is NULL, could not gather data\n");
147
148         if(numScanPoints > 0)
149             terrain->AddPointsToWorld(scannedData, numScanPoints);
150         else if(numScanPoints < 0)
151             printf("Error - integer representing the number of points gathered,
                    was negative\n");
152
153         free(scannedData);
154         scannedData = NULL;
155
156         terrain->ProcessPoints(KNOWN);
157     }
158
159     /** Version for testing purposes only
160      */
161     void Robot::GatherData(Coord* scannedData, int numScanPoints)
162     {
163         if(scannedData != NULL && numScanPoints != 0)
164             terrain->AddPointsToWorld(scannedData, numScanPoints);
165         terrain->ProcessPoints(KNOWN);
166     }
167
168     /** One of the two main functions of robot.
169      * This calls the functions to get and process/prepare data
170      */
171     void Robot::GatherData(SimulationEnvironment* simEnv, int inputQualityLevel)
            // could this just use already stored sim?
172     //perhaps use inputLevel for number of scanPoint in x and y axis?
173     {
174         Coord* scannedData;
175         int numScanPoints = 0;
176         if(sim < RESERVED)
177             printf("Error - Sim is NULL, could not gather data\n");
178
179 //      int size = 121;
180 //      int size = 231;
181         int size = 441;
182         scannedData = (Coord*) calloc(size, sizeof(Coord));
183         sim->ScanViewPoints(&scannedData, &numScanPoints, currentPos,
                directionFacing);
184         if(numScanPoints < 0)
185                 printf("Error - integer representing the number of points
                        gathered, was negative\n");
186         else
187         {
```

```
188             if(numScanPoints > 0)
189                 terrain->AddPointsToWorld(scannedData, numScanPoints);
190         }
191         free(scannedData);
192         scannedData = NULL;
193         terrain->ProcessPoints(KNOWN);
194     }
195
196     /** Works out which edge of the triangle testPos is in, would be crossed in
            getting to it from current pos, so can use the vector of that edge for
            new movement direction to avoid obstacle
197      */
198     Coord Robot::CheckEdgePath(Coord start, Coord* testPos, int numPoints, Coord*
            pastTestPoints)
199     {
200         bool loops = false;
201         for(int i = 0; i < numPoints - 1; i++)
202         {
203             if((pastTestPoints[i]) == *testPos)
204             {
205                 loops = true;
206                 break;
207             }
208         }
209
210         Triangle* encompTri = new Triangle;
211         Coord edgeVector, newPos;
212         bool vertCrossed = false;
213         bool triExists = false;
214         for(int i = 0; i < 3 && !triExists && !loops; i++)
215         {
216             if(terrain->GetTri(&encompTri, &start))
217             {
218                 if(encompTri->hazardType)
219                 {
220                     break;
221                 }
222
223                 int triTestRes = InsideTriangleTest(start, *encompTri->point1, *
                        encompTri->point2, *encompTri->point3);
224                 if( triTestRes == ON_EDGE || triTestRes == ON_VERTEX ||
                        triTestRes == ABOVE_EDGE)
225                 {
226                     if(i == 0)
227                         start = start + NormaliseVector(*testPos - start, 5.0f,
                                false);//or minus last move vector
228                     else if(i == 1)
```

```
229                     {
230                         Coord normVector = CrossProduct(*testPos - start, *
                                testPos - start + Coord(0,0,10));
231                         start = start + NormaliseVector(normVector, 5.0f, false);
                                //or minus last move vector
232                     }
233                     else
234                     {
235                         Coord centre = CalcTriCentre(*encompTri->point1, *
                                encompTri->point2, *encompTri->point3);
236                         start = start + NormaliseVector(centre - start, 5.0f,
                                false);//or minus last move vector
237                         triExists = true;
238                     }
239                 }
240                 else
241                     triExists = true;
242             }
243             else
244                 break;
245         }
246 
247         if(triExists)
248         {
249             if(CheckEdgeIntersections(true, false, *encompTri->GetPoint(0), *
                    encompTri->GetPoint(1), *testPos, start, NULL))
250                 edgeVector = *encompTri->GetPoint(0) - *encompTri->GetPoint(1);
251             else if(CheckEdgeIntersections(true, false, *encompTri->GetPoint(0),
                    *encompTri->GetPoint(2), *testPos, start, NULL))
252                 edgeVector = *encompTri->GetPoint(0) - *encompTri->GetPoint(2);
253             else if(CheckEdgeIntersections(true, false, *encompTri->GetPoint(1),
                    *encompTri->GetPoint(2), *testPos, start, NULL))
254                 edgeVector = *encompTri->GetPoint(1) - *encompTri->GetPoint(2);
255             else if(InsideTriangleTest(*testPos, *encompTri->GetPoint(0), *
                    encompTri->GetPoint(1), *encompTri->GetPoint(2)))
256             {
257                 printf("Error - Current position is inside Triangle being tested\
                        n");
258                 edgeVector = Coord(0,0,0);
259             }
260             else
261             {
262                 Coord directionVector = *testPos - start;
263                 float moveDist = CLEARANCEBUFFER_MM;
264                 directionVector = NormaliseVector<Coord>(directionVector,
                        moveDist, false);
```

```
265                 Coord normVector = CrossProduct(directionVector, directionVector
                        + Coord(0,0,10));
266                 pastTestPoints[numPoints++] = (*testPos + NormaliseVector<Coord>(
                        normVector, 2.0f, false));
267                 newPos = CheckEdgePath(start, &(*testPos + NormaliseVector<Coord
                        >(normVector, 2.0f, false)), numPoints, pastTestPoints);
268                 vertCrossed = true;
269             }
270
271             if(!vertCrossed)//as when vert cross new pos is all sorted
272             {
273                 edgeVector = NormaliseVector<Coord>(edgeVector,
                        CLEARANCEBUFFER_MM, false);
274                 if(edgeVector.xVal * (goal.yVal - currentPos.yVal) - (goal.xVal -
                          currentPos.xVal) * edgeVector.yVal > 0)
275                     newPos = currentPos + edgeVector;
276                 else
277                     newPos = currentPos - edgeVector;
278
279                 pastTestPoints[numPoints++] = newPos;
280                 if(terrain->GetTri(&encompTri, &newPos) && encompTri->hazardType)
281                     newPos = CheckEdgePath(start, &newPos, numPoints,
                            pastTestPoints);
282             }
283         }
284
285         delete encompTri;
286
287         return newPos;
288     }
289
290     /** Second of the two main functions of robot.
291      * This calls/controls the functions which analyse the terrain and plan a
             path
292      */
293     void Robot::PlanPath()
294     {
295         switch(algorithm)
296         {
297             case NO_ALG:
298                 break;
299             case HUGGER:
300             {
301                 algorithmUsed = HUGGER;
302                     EdgeHugger<Face_handle, Point>();
303                 break;
304             }
```

```
305             case DFS:
306             {
307                 algorithmUsed = DFS;
308                 DepthFirst<ALG_NODE_TYPE>(this);
309                 break;
310             }
311             case BFS:
312             {
313                 algorithmUsed = BFS;
314                 BreadthFirst<ALG_NODE_TYPE>(this);
315                 break;
316             }
317             case GREEDY:
318             {
319                 algorithmUsed = GREEDY;
320                 GreedyPaths<ALG_NODE_TYPE>(this);
321                 break;
322             }
323             case STATE:
324             {
325                 algorithmUsed = STATE;
326                 StateAlg<ALG_NODE_TYPE>(this);
327                 break;
328             }
329             case ASTAR:
330             {
331                 algorithmUsed = ASTAR;
332                 A_Star_Ctrl<ALG_NODE_TYPE>(this);
333                 break;
334             }
335             case LPASTAR:
336             {
337                 algorithmUsed = LPASTAR;
338                 LPA_Star<ALG_NODE_TYPE>(this);
339                 break;
340             }
341 //          case DSTAR:
342 //          {
343 //              algorithmUsed = DSTAR;
344 //              D_Star(this);
345 //              break;
346 //          }
347
348             default:
349                 break;
350         }
351     }
```

```
352
353     /** Used to indicate whether the robot should continue
354      * Checks to see if it has reached its destination and if not whether
355      * it's possible to reach the destination.
356      */
357     bool Robot::NaviDone()
358     {
359         if(CalculateDirectDistance(currentPos, goal, true) > 2 && terrain->
                GoalReachable())
360             return false;
361         else
362             return true;
363     }
364
365     /** Calculate the vector/direction of movement between current and next point
366      */
367     Coord Robot::MoveDirection()
368     {
369         Point moveCoords = Point(0,0,0), temp;
370         switch(algorithmUsed)
371         {
372             case NO_ALG:
373                 break;
374             case HUGGER ... EIGHT:
375             {
376                 if(numStepsInPath == 1)
377                 {
378                     temp = plannedPath[0];
379                     moveCoords = temp - (currentPt - ZERO_POINT);
380                 }
381                 else
382                     printf("Error: ********* No next move *********\n");
383
384                 break;
385             }
386             case DFS ... DSTAR:
387             {
388                 if(numStepsInPath > 1)
389                 {
390                     temp = plannedPath[1];
391                     moveCoords = temp - (currentPt - ZERO_POINT);
392                     RecordMoreData("Way_Points.txt", temp.x(), temp.y(), temp.z()
                            );
393                 }
394                 else
395                     printf("Error: ********* No next move *********\n");
396             }
```

```
397             default:
398                 break;
399         }
400         return ConvertPointType<Point, Coord>(moveCoords);
401     }
402
403     void Robot::SetPath(Point* path)
404     {
405         plannedPath = (Point*) realloc(plannedPath, numStepsInPath * sizeof(Point
                ));
406         for(int i = 0; i < numStepsInPath; i++)
407             plannedPath[i] = path[i];
408
409         if(algorithmUsed == EIGHT)
410             free(path);
411     }
412
413     void Robot::SetNumSteps(int steps)
414     {
415         numStepsInPath = steps;
416     }
417
418     void Robot::SetDirection(Coord direction)
419     {
420         directionFacing = direction;
421     }
422
423     Coord Robot::GetDirection()
424     {
425         return directionFacing;
426     }
427
428     Coord Robot::GetPosition()
429     {
430         return currentPos;
431     }
432
433     Point Robot::GetNext()
434     {
435         if(numStepsInPath > 1)
436             return plannedPath[1];
437         else
438             return Point(0,0,0);
439     }
440
441     Point Robot::GetGoalPt()
442     {
```

```
443         return goalPt;
444     }
445
446     Point Robot::GetStartPt()
447     {
448         return startPt;
449     }
450
451     Coord Robot::GetGoal()
452     {
453         return goal;
454     }
455
456     Coord Robot::GetLastMove()
457     {
458         return lastMove;
459     }
460
461     /** Update the position the robot believes itself to be
462      */
463     void Robot::AdjustPosition(Coord change)
464     {
465         if(XYMatch(lastMove, Coord(-change.x(), -change.y(), -change.z())))
466         {
467             if(consecutiveTimesAlternating++ == 10)
468                 sim->EndRun(false, "Robot was stuck alternating between two
                        positions. \t");
469         }
470         else
471         {
472             consecutiveTimesAlternating = 0;
473         }
474
475         currentPos = currentPos + change;
476         currentPt = currentPt + (ConvertPointType<Coord, Point>(change) -
                ZERO_POINT);
477         terrain->AdjustLocation(change.x(), change.y(), change.z());
478         lastMove = change;
479     }
480
481     /** Update the position the robot believes itself to be
482      */
483     void Robot::AdjustPosition(int xChange, int yChange, int zChange)
484     {
485         if(XYMatch(lastMove, Coord(-xChange, -yChange, -zChange)))
486         {
487             if(consecutiveTimesAlternating++ == 10)
```

```
488                 sim->EndRun(false, "Robot was stuck alternating between two
                        positions. \t");
489         }
490         else
491         {
492             consecutiveTimesAlternating = 0;
493         }
494 
495         currentPos.xVal += xChange;
496         currentPos.yVal += yChange;
497         currentPos.zVal += zChange;
498         currentPt = currentPt + (Point(xChange, yChange, zChange) - ZERO_POINT);
499         terrain->AdjustLocation(xChange, yChange, zChange);
500         lastMove = Coord(xChange, yChange, zChange);
501     }
```

A.7 SimulationEnvironment

A.7.1 SimulationEnvironment.h

```
1  /*************************************************************************
2   *   Copyright (C) 2007 by Michael Douglas Hasler   *
3   *   redheadpunk@gmail.com   *
4   *                                                                       *
5   *   This program is free software; you can redistribute it and/or modify *
6   *   it under the terms of the GNU General Public License as published by *
7   *   the Free Software Foundation; either version 2 of the License, or    *
8   *   (at your option) any later version.                                  *
9   *                                                                       *
10  *   This program is distributed in the hope that it will be useful,      *
11  *   but WITHOUT ANY WARRANTY; without even the implied warranty of       *
12  *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the        *
13  *   GNU General Public License for more details.                         *
14  *                                                                       *
15  *   You should have received a copy of the GNU General Public License    *
16  *   along with this program; if not, write to the                        *
17  *   Free Software Foundation, Inc.,                                      *
18  *   59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.            *
19  *************************************************************************/
20 
21 /**
22 Potential things to do:
23     Overlay which shows where robot has been, where is looking
24     Relativity of coords ie relative to world, map, gps etc
25 */
```

```
26
27 /**
28 Notes:
29     Values in the sim are in centimetres
30     Values held by robot & world are in milimetres
31 */
32
33 #ifndef SIMULATION_ENVIRONMENT_H_
34 #define SIMULATION_ENVIRONMENT_H_
35
36 #ifndef IRRLICHT_H_
37 #define IRRLICHT_H_
38 #include <irrlicht/irrlicht.h>
39 #endif
40
41 #include <math.h>
42 #include <iostream>
43 #include <string>
44
45 #define INPUT_TYPE 1
46
47 class SimulationEnvironment;
48 #include "Robot.h"
49
50 using namespace irr;
51 using namespace gui;
52 #pragma comment(lib, "Irrlicht.lib")
53
54 const bool SHOW_SIM_ERRORS = true;
55 const bool SHOW_SIM_WARNINGS = false;
56
57 const int WINDOW_WIDTH = 640;
58 const int VIEW_HEIGHT = 480;
59 const int WINDOW_HEIGHT = 640;
60 const int GAP = 10;
61 const int LINE_HEIGHT = 18;
62 const int COLUMN_WIDTH = 200;
63
64 const int ROBOTNODE_VERTBOOST = 50;
65 const int CAMERA_VERTBOOST = 50;
66 const int CHAR_ARRAY_CHUNK = 255;
67
68 const bool INCLUDE_NOISE = false;
69 const float MAX_NOISE  = 0.1;
70
71 enum KEY_MODES{
72     NO_MODE = 0,
```

```
73      SIM_MODE,
74      ROBOT_MODE,
75      DISPLAY_MODE
76      };
77
78  enum GUI_COMPONENT{
79      GUI_ID_MODE = 101,
80      GUI_ID_CTRLS,
81      GUI_ID_LAST_MOVE,
82      GUI_ID_TRANSPARENCY_SCROLL_BAR,
83      GUI_ACT_WIN_MODE
84      };
85
86  struct SimConfig
87  {
88      int algorithm;
89      char *heightMap, *textureOverlay, *closeTextureOverlay;
90      core::vector3df startPosition, goalPosition, cameraPositions[3],
            cameraTargets[3];
91      float cameraFarValue[3];
92  };
93
94  struct ObstacleConfig
95  {
96      char *mesh, *textureOverlay;
97      core::vector3df position, rotation, scale;
98  };
99
100 class MyEventReceiver : public IEventReceiver
101 {
102 public:
103     MyEventReceiver(scene::ISceneNode* terrain, scene::ISceneManager* manager,
            gui::IGUIEnvironment* environ, SimulationEnvironment* sim) :
104         terrain(terrain), simEnviron(sim), sManager(manager), gEnv(environ),
                mode(NO_MODE) { }
105     bool OnEvent(const SEvent& event);
106 private:
107     scene::ISceneNode* terrain;
108     SimulationEnvironment* simEnviron;
109     gui::IGUIEnvironment* gEnv;
110     scene::ISceneManager* sManager;
111     KEY_MODES mode;
112 };
113
114 class SimulationEnvironment
115 {
116 public:
```

```
117     SimulationEnvironment();
118     SimulationEnvironment(char* configFilePath, char* obstaclesFilePath);
119     ~SimulationEnvironment();
120     int SetupEnvironment(char* configFilePath, char* obstaclesFilePath);
121     void RunEnvironment();
122     void RunReplay(char* inputFilePath, int numIter, int moveSize);
123
124     void const SwitchView();
125     void const SwitchView(int num);
126     void ToggleSimRun();
127     void ToggleRobotRun();
128     void ToggleMinMax();
129     void ToggleActiveWindowMode();
130     void ToggleSkies();
131
132     void DisplayGraphLinks();
133     void DisplayPath();
134
135     void ScanViewPoints(Coord** terrainPoints, int *numPoints, Coord position,
            Coord direction);
136     void ScanViewPoints_v2(Coord** terrainPoints, int *numPoints);
137     void PanView();
138     bool Exited();
139     void ExitSim();
140     void EndRun();
141     void EndRun(bool goalReached, char* message);
142
143 private:
144     void SetupGui();
145     void UpdateGui();
146     void SetupRobot();
147     bool UpdateRobot();
148
149     void CloseEnvironment();
150
151     void ImportSettingsConfig(char* configFilePathm);
152     void ImportObstaclesConfig(char* obstaclesFilePath, ObstacleConfig** oConfig,
            int* numObstacles);
153     void AddObstacles(ObstacleConfig* obstacles, int numObstacles);
154
155     float CheckForCollision(int start[3], int end[3]);
156     float CheckForCollision(core::vector3df start);
157
158     scene::ITerrainSceneNode* GetTerrain();
159     IrrlichtDevice* GetDevice();
160     scene::ICameraSceneNode* GetCamera(int num);
161
```

```
162     bool IsViewable(core::vector3df point1, core::vector3df point2, core::
            vector3df point3);
163     bool CheckIfDone();
164
165     IrrlichtDevice* device;
166     gui::IGUIEnvironment* env;
167     scene::IBillboardSceneNode * bill;
168     scene::ICameraSceneNode** camera;
169     scene::ISceneManager* smgr;
170     scene::ISceneNode* robotNode;
171     scene::ISceneNode* skybox;
172     scene::ISceneNode* skydome;
173     int numAnim;
174     scene::ISceneNodeAnimatorCollisionResponse** anim;
175     scene::ITerrainSceneNode* terrain;
176     scene::IMetaTriangleSelector* metaSelector;
177     video::E_DRIVER_TYPE driverType;
178     video::IVideoDriver* driver;
179
180     int view;
181     bool showBox;
182     bool runSim;
183     bool runRobot;
184     bool allowInactiveScreen;
185     SimConfig sConfig;
186
187     Robot* robot;
188 };
189
190 #endif /*MAIN_H_*/
191
192 //  KEY_LBUTTON          = 0x01,  // Left mouse button
193 //  KEY_RBUTTON          = 0x02,  // Right mouse button
194 //  KEY_CANCEL           = 0x03,  // Control-break processing
195 //  KEY_MBUTTON          = 0x04,  // Middle mouse button (three-button mouse)
196 //  KEY_XBUTTON1         = 0x05,  // Windows 2000/XP: X1 mouse button
197 //  KEY_XBUTTON2         = 0x06,  // Windows 2000/XP: X2 mouse button
198 //  KEY_BACK             = 0x08,  // BACKSPACE key
199 //  KEY_TAB              = 0x09,  // TAB key
200 //  KEY_CLEAR            = 0x0C,  // CLEAR key
201 //  KEY_RETURN           = 0x0D,  // ENTER key
202 //  KEY_SHIFT            = 0x10,  // SHIFT key
203 //  KEY_CONTROL          = 0x11,  // CTRL key
204 //  KEY_MENU             = 0x12,  // ALT key
205 //  KEY_PAUSE            = 0x13,  // PAUSE key
206 //  KEY_CAPITAL          = 0x14,  // CAPS LOCK key
207 //  KEY_ESCAPE           = 0x1B,  // ESC key
```

```
208 //  KEY_SPACE          = 0x20,  // SPACEBAR
209 //  KEY_PRIOR          = 0x21,  // PAGE UP key
210 //  KEY_NEXT           = 0x22,  // PAGE DOWN key
211 //  KEY_END            = 0x23,  // END key
212 //  KEY_HOME           = 0x24,  // HOME key
213 //  KEY_LEFT           = 0x25,  // LEFT ARROW key
214 //  KEY_UP             = 0x26,  // UP ARROW key
215 //  KEY_RIGHT          = 0x27,  // RIGHT ARROW key
216 //  KEY_DOWN           = 0x28,  // DOWN ARROW key
217 //  KEY_SNAPSHOT       = 0x2C,  // PRINT SCREEN key
218 //  KEY_INSERT         = 0x2D,  // INS key
219 //  KEY_DELETE         = 0x2E,  // DEL key
220 //  KEY_KEY_0          = 0x30,  // 0 key
221 //  KEY_KEY_1          = 0x31,  // 1 key
222 //  KEY_KEY_2          = 0x32,  // 2 key
223 //  KEY_KEY_3          = 0x33,  // 3 key
224 //  KEY_KEY_4          = 0x34,  // 4 key
225 //  KEY_KEY_5          = 0x35,  // 5 key
226 //  KEY_KEY_6          = 0x36,  // 6 key
227 //  KEY_KEY_7          = 0x37,  // 7 key
228 //  KEY_KEY_8          = 0x38,  // 8 key
229 //  KEY_KEY_9          = 0x39,  // 9 key
230 //  KEY_KEY_A          = 0x41,  // A key
231 //  KEY_KEY_B          = 0x42,  // B key
232 //  KEY_KEY_C          = 0x43,  // C key
233 //  KEY_KEY_D          = 0x44,  // D key
234 //  KEY_KEY_E          = 0x45,  // E key
235 //  KEY_KEY_F          = 0x46,  // F key
236 //  KEY_KEY_G          = 0x47,  // G key
237 //  KEY_KEY_H          = 0x48,  // H key
238 //  KEY_KEY_I          = 0x49,  // I key
239 //  KEY_KEY_J          = 0x4A,  // J key
240 //  KEY_KEY_K          = 0x4B,  // K key
241 //  KEY_KEY_L          = 0x4C,  // L key
242 //  KEY_KEY_M          = 0x4D,  // M key
243 //  KEY_KEY_N          = 0x4E,  // N key
244 //  KEY_KEY_O          = 0x4F,  // O key
245 //  KEY_KEY_P          = 0x50,  // P key
246 //  KEY_KEY_Q          = 0x51,  // Q key
247 //  KEY_KEY_R          = 0x52,  // R key
248 //  KEY_KEY_S          = 0x53,  // S key
249 //  KEY_KEY_T          = 0x54,  // T key
250 //  KEY_KEY_U          = 0x55,  // U key
251 //  KEY_KEY_V          = 0x56,  // V key
252 //  KEY_KEY_W          = 0x57,  // W key
253 //  KEY_KEY_X          = 0x58,  // X key
254 //  KEY_KEY_Y          = 0x59,  // Y key
```

```
255 //   KEY_KEY_Z          = 0x5A,   // Z key
256 //   KEY_LWIN           = 0x5B,   // Left Windows key (Microsoft Natural
        keyboard)
257 //   KEY_RWIN           = 0x5C,   // Right Windows key (Natural keyboard)
258 //   KEY_SLEEP          = 0x5F,   // Computer Sleep key
259 //   KEY_NUMPAD0        = 0x60,   // Numeric keypad 0 key
260 //   KEY_NUMPAD1        = 0x61,   // Numeric keypad 1 key
261 //   KEY_NUMPAD2        = 0x62,   // Numeric keypad 2 key
262 //   KEY_NUMPAD3        = 0x63,   // Numeric keypad 3 key
263 //   KEY_NUMPAD4        = 0x64,   // Numeric keypad 4 key
264 //   KEY_NUMPAD5        = 0x65,   // Numeric keypad 5 key
265 //   KEY_NUMPAD6        = 0x66,   // Numeric keypad 6 key
266 //   KEY_NUMPAD7        = 0x67,   // Numeric keypad 7 key
267 //   KEY_NUMPAD8        = 0x68,   // Numeric keypad 8 key
268 //   KEY_NUMPAD9        = 0x69,   // Numeric keypad 9 key
269 //   KEY_MULTIPLY       = 0x6A,   // Multiply key
270 //   KEY_ADD            = 0x6B,   // Add key
271 //   KEY_SEPARATOR      = 0x6C,   // Separator key
272 //   KEY_SUBTRACT       = 0x6D,   // Subtract key
273 //   KEY_DECIMAL        = 0x6E,   // Decimal key
274 //   KEY_DIVIDE         = 0x6F,   // Divide key
275 //   KEY_F1             = 0x70,   // F1 key
276 //   KEY_F2             = 0x71,   // F2 key
277 //   KEY_F3             = 0x72,   // F3 key
278 //   KEY_F4             = 0x73,   // F4 key
279 //   KEY_F5             = 0x74,   // F5 key
280 //   KEY_F6             = 0x75,   // F6 key
281 //   KEY_F7             = 0x76,   // F7 key
282 //   KEY_F8             = 0x77,   // F8 key
283 //   KEY_F9             = 0x78,   // F9 key
284 //   KEY_F10            = 0x79,   // F10 key
285 //   KEY_F11            = 0x7A,   // F11 key
286 //   KEY_F12            = 0x7B,   // F12 key
287 //   KEY_NUMLOCK        = 0x90,   // NUM LOCK key
288 //   KEY_SCROLL         = 0x91,   // SCROLL LOCK key
289 //   KEY_LSHIFT         = 0xA0,   // Left SHIFT key
290 //   KEY_RSHIFT         = 0xA1,   // Right SHIFT key
291 //   KEY_LCONTROL       = 0xA2,   // Left CONTROL key
292 //   KEY_RCONTROL       = 0xA3,   // Right CONTROL key
293 //   KEY_LMENU          = 0xA4,   // Left MENU key
294 //   KEY_RMENU          = 0xA5,   // Right MENU key
295 //   KEY_PLUS           = 0xBB,   // Plus Key   (+)
296 //   KEY_COMMA          = 0xBC,   // Comma Key  (,)
297 //   KEY_MINUS          = 0xBD,   // Minus Key  (-)
298 //   KEY_PERIOD         = 0xBE,   // Period Key (.)
299 //
```

```
300 //  KEY_KEY_CODES_COUNT  = 0xFF // this is not a key, but the amount of keycodes
        there are.
```

A.7.2 SimEnv.cpp

```
1  /**********************************************************************
2   *   Copyright (C) 2009 by Michael Douglas Hasler   *
3   *   redheadpunk@gmail.com   *
4   *                                                                    *
5   *   This program is free software; you can redistribute it and/or modify  *
6   *   it under the terms of the GNU General Public License as published by  *
7   *   the Free Software Foundation; either version 2 of the License, or     *
8   *   (at your option) any later version.                                   *
9   *                                                                    *
10  *   This program is distributed in the hope that it will be useful,       *
11  *   but WITHOUT ANY WARRANTY; without even the implied warranty of        *
12  *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the         *
13  *   GNU General Public License for more details.                          *
14  *                                                                    *
15  *   You should have received a copy of the GNU General Public License     *
16  *   along with this program; if not, write to the                         *
17  *   Free Software Foundation, Inc.,                                       *
18  *   59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.             *
19  **********************************************************************/
20 #ifndef IRRLICHT_H_
21 #define IRRLICHT_H_
22 #include <irrlicht/irrlicht.h>
23 #endif
24
25 #include <iostream>
26 #include <string>
27 #include <stdio.h>
28
29 #include "adv_math.h"
30 #include "SimulationEnvironment.h"
31
32 using namespace irr;
33
34 #ifdef _MSC_VER
35 #pragma comment(lib, "Irrlicht.lib")
36 #endif
37
38 #define DEBUG 1
39
40     bool MyEventReceiver::OnEvent(const SEvent& event)
41     {
```

```
42          bool result = false;
43          if (event.EventType == irr::EET_KEY_INPUT_EVENT && !event.KeyInput.
                PressedDown)
44          {
45              result = true;
46              switch (event.KeyInput.Key)
47              {
48                  case irr::KEY_PAUSE: // pause or continue sim
49                      simEnviron->ToggleSimRun();
50                      break;
51                  case irr::KEY_ESCAPE:
52                  {
53                      mode = NO_MODE;
54
55                      gui::IGUIElement* rootElement = gEnv->getRootGUIElement();
56
57                      rootElement->removeChild(rootElement->getElementFromId(
                            GUI_ID_MODE, false));
58                      gEnv->addStaticText(L"No Mode Selected", core::rect<s32>(GAP
                            + 45 , VIEW_HEIGHT + GAP, GAP + COLUMN_WIDTH, VIEW_HEIGHT
                             + GAP + LINE_HEIGHT), true, true, 0, GUI_ID_MODE, true);
59
60                      rootElement->removeChild(rootElement->getElementFromId(
                            GUI_ID_CTRLS, false));
61                      gEnv->addStaticText(L"Press:\n '1' to see robot's view\n '2'
                            for user controlled camera\n '3' for bird's eye view\n 'C
                            ' to close the simulation", core::rect<s32>(GAP,
                            VIEW_HEIGHT + GAP + LINE_HEIGHT, GAP + COLUMN_WIDTH,
                            WINDOW_HEIGHT - GAP), true, true, 0, GUI_ID_CTRLS, true);
62                      break;
63                  }
64                  case irr::KEY_KEY_1:
65                      simEnviron->SwitchView(0);
66                      break;
67                  case irr::KEY_KEY_2:
68                      simEnviron->SwitchView(1);
69                      break;
70                  case irr::KEY_KEY_3:
71                      simEnviron->SwitchView(2);
72                      break;
73                  case irr::KEY_KEY_C: // close simulation
74                      simEnviron->ExitSim();
75                      break;
76                  case irr::KEY_KEY_D:
77                  {
78                      if(mode == SIM_MODE)
79                          terrain->setMaterialType(
```

```
80                      terrain->getMaterial(0).MaterialType == video::
                            EMT_SOLID ?
81                      video::EMT_DETAIL_MAP : video::EMT_SOLID);
82                  else if(mode == DISPLAY_MODE)
83                      simEnviron->DisplayPath();
84                  break;
85              }
86              case irr::KEY_KEY_G:
87                  if(mode == DISPLAY_MODE)
88                      simEnviron->DisplayGraphLinks();
89                  break;
90              case irr::KEY_KEY_H:
91                  if(mode == DISPLAY_MODE)
92                      display("hazards.txt");
93                  break;
94              case irr::KEY_KEY_I: // toggle whether sim runs while window is
                    not active
95                  simEnviron->ToggleActiveWindowMode();
96                  break;
97              case irr::KEY_KEY_L:
98                  if(mode == DISPLAY_MODE)
99                      display("links.txt");
100                 break;
101             case irr::KEY_KEY_M:
102                 simEnviron->ToggleMinMax();
103                 break;
104             case irr::KEY_KEY_P:
105             {
106                 if(mode == SIM_MODE) // switch point frame mode
107                 {
108                     terrain->setMaterialFlag(video::EMF_POINTCLOUD, !terrain
                            ->getMaterial(0).PointCloud);
109                     terrain->setMaterialFlag(video::EMF_WIREFRAME, false);
110                 }
111                 else if(mode == ROBOT_MODE)
112                     simEnviron->ToggleRobotRun();
113                 break;
114             }
115             case irr::KEY_KEY_S: // toggle skies
116                 if(mode == SIM_MODE)
117                     simEnviron->ToggleSkies();
118                 else if(mode == ROBOT_MODE)
119                     simEnviron->EndRun();
120                 break;
121             case irr::KEY_KEY_T: // display tessellation
122                 if(mode == DISPLAY_MODE)
123                     display("mesh_triangles.txt");
```

```
124                     break;
125                 case irr::KEY_KEY_W:
126                 {
127                     if(mode == SIM_MODE)  // switch wire frame mode
128                     {
129                         terrain->setMaterialFlag(video::EMF_WIREFRAME, !terrain->
                                getMaterial(0).Wireframe);
130                         terrain->setMaterialFlag(video::EMF_POINTCLOUD, false);
131                     }
132                     break;
133                 }
134                 case irr::KEY_F1:
135                 {
136                     mode = SIM_MODE;
137                     gui::IGUIElement* rootElement = gEnv->getRootGUIElement();
138 
139                     rootElement->removeChild(rootElement->getElementFromId(
                            GUI_ID_MODE, false));
140                     gEnv->addStaticText(L"Sim Ctrl Mode", core::rect<s32>(GAP +
                            45 , VIEW_HEIGHT + GAP, GAP + COLUMN_WIDTH, VIEW_HEIGHT +
                             GAP + LINE_HEIGHT), true, true, 0, GUI_ID_MODE, true);
141 
142                     rootElement->removeChild(rootElement->getElementFromId(
                            GUI_ID_CTRLS, false));
143                     gEnv->addStaticText(L"Press:\n 'D' to toggle detail map\n 'P'
                            to toggle point cloud mode\n 'S' to toggle skybox/
                            skydome\n 'W' to change wireframe mode", core::rect<s32>(
                            GAP, VIEW_HEIGHT + GAP + LINE_HEIGHT, GAP + COLUMN_WIDTH,
                             WINDOW_HEIGHT - GAP), true, true, 0, GUI_ID_CTRLS, true)
                            ;
144                     break;
145                 }
146                 case irr::KEY_F2:
147                 {
148                     mode = ROBOT_MODE;
149                     gui::IGUIElement* rootElement = gEnv->getRootGUIElement();
150 
151                     rootElement->removeChild(rootElement->getElementFromId(
                            GUI_ID_MODE, false));
152                     gEnv->addStaticText(L"Robot Ctrl Mode", core::rect<s32>(GAP +
                             45 , VIEW_HEIGHT + GAP, GAP + COLUMN_WIDTH, VIEW_HEIGHT
                            + GAP + LINE_HEIGHT), true, true,0, GUI_ID_MODE, true);
153 
154                     rootElement->removeChild(rootElement->getElementFromId(
                            GUI_ID_CTRLS, false));
155                     gEnv->addStaticText(L"Press:\n 'P' to Pause/Run Robot\n 'S'
                            to skip to next run", core::rect<s32>(GAP, VIEW_HEIGHT +
```

```
                        GAP + LINE_HEIGHT, GAP + COLUMN_WIDTH, WINDOW_HEIGHT -
                        GAP), true, true, 0, GUI_ID_CTRLS, true);
156                 break;
157             }
158             case irr::KEY_F3:
159             {
160                 mode = DISPLAY_MODE;
161                 gui::IGUIElement* rootElement = gEnv->getRootGUIElement();
162
163                 rootElement->removeChild(rootElement->getElementFromId(
                        GUI_ID_MODE, false));
164                 gEnv->addStaticText(L"Display Ctrl Mode", core::rect<s32>(GAP
                         + 45 , VIEW_HEIGHT + GAP, GAP + COLUMN_WIDTH,
                        VIEW_HEIGHT + GAP + LINE_HEIGHT), true, true, 0,
                        GUI_ID_MODE, true);
165
166                 rootElement->removeChild(rootElement->getElementFromId(
                        GUI_ID_CTRLS, false));
167                 gEnv->addStaticText(L"Press:\n 'D' to display planned path\n
                        'G' to display all links\n 'H' to display hazards\n 'L'
                        to display links\n 'T' to display tessellation", core::
                        rect<s32>(GAP, VIEW_HEIGHT + GAP + LINE_HEIGHT, GAP +
                        COLUMN_WIDTH, WINDOW_HEIGHT - GAP), true, true, 0,
                        GUI_ID_CTRLS, true);
168                 break;
169             }
170             default:
171                 result = false;
172         }
173     }
174
175     return result;
176 }
177
178 int ReadInFilenames(char configFilePath[100], char*** simConfigFilePath, char
        *** obstaclesConfigFilePath)
179 {
180     FILE* configFile;
181     configFile = fopen(configFilePath, "rt");
182     if (configFile == NULL)
183     {
184         if(SHOW_SIM_ERRORS) printf("Error - Unable to open file containing
                configurations\n");
185         exit(1); // terminate with error
186     }
187
188     char buffer[255];
```

```
189         char simConfig[255];
190         char obstConfig[255];
191         int numConfigs = 0;
192         while(fgets(buffer, 255, configFile) != NULL)
193         {
194             if(buffer[0] != '\n')
195                 numConfigs++;
196         }
197 
198         if(numConfigs == 0)
199         {
200             *simConfigFilePath = (char**) calloc(1, sizeof(char*));
201             *obstaclesConfigFilePath = (char**) calloc(1, sizeof(char*));
202         }
203         else
204         {
205             *simConfigFilePath = (char**) realloc(*simConfigFilePath, numConfigs
                    * sizeof(char*));
206             *obstaclesConfigFilePath = (char**) realloc(*obstaclesConfigFilePath,
                     numConfigs * sizeof(char*));
207         }
208 
209         rewind(configFile);
210         for(int i = 0; i < numConfigs; i++)
211         {
212             (*simConfigFilePath)[i] = (char*) calloc(255, sizeof(char));
213             (*obstaclesConfigFilePath)[i] = (char*) calloc(255, sizeof(char));
214             fgets(buffer, 255, configFile);
215             sscanf(buffer, "%s %s", (*simConfigFilePath)[i], (*
                    obstaclesConfigFilePath)[i]);
216             if((*simConfigFilePath)[i][0] == '\n' || (*obstaclesConfigFilePath)[i
                    ][0] == '\n')
217             {
218                 free((*simConfigFilePath)[i]);
219                 free((*obstaclesConfigFilePath)[i]);
220                 i--;
221                 continue;
222             }
223         }
224 
225         return numConfigs;
226     }
227 
228     void runSim(int argc, char* argv[])
229     {
230         int numRuns = 1;
231         int numConfigs = 1;
```

```
232         char configFilePath[100];
233         char** simConfigFilePath;
234         char** obstaclesConfigFilePath;
235         sscanf(argv[0], "%s", configFilePath);
236         simConfigFilePath = (char**) calloc(1, sizeof(char*));
237         obstaclesConfigFilePath = (char**) calloc(1, sizeof(char*));
238         numConfigs = ReadInFilenames(configFilePath, &simConfigFilePath, &
                obstaclesConfigFilePath);
239
240         if(numConfigs == 0)
241         {
242             numConfigs = 1;
243             sscanf("/home/haz/SimEnv/sim.config\n", "%s", simConfigFilePath[0]);
244             sscanf("/home/haz/SimEnv/no_obstacles.config\n", "%s",
                    obstaclesConfigFilePath[0]);
245             printf("No valid file paths for config files were found in given
                    input file, defaults being used\n");
246         }
247
248         if(argc > 1)
249             sscanf(argv[1], "%d", &numRuns);
250
251         SimulationEnvironment simEnv = SimulationEnvironment();
252         int i = 0;
253         for(i = 0; i < numConfigs && !simEnv.Exited(); i++)
254         {
255             for(int j = 0; j < numRuns && !simEnv.Exited(); j++)
256             {
257                 if(simEnv.SetupEnvironment(simConfigFilePath[i],
                        obstaclesConfigFilePath[i]) == 1)
258                     break;
259                 simEnv.RunEnvironment();
260             }
261         }
262
263         for(int i = 0; i < numConfigs; i++)
264         {
265             free(simConfigFilePath[i]);
266             free(obstaclesConfigFilePath[i]);
267         }
268
269         free(simConfigFilePath);
270         free(obstaclesConfigFilePath);
271     }
272
273     void recreateTraversedPath(char* filename, int numIterations, int moveSize)
274     {
```

```
275         int chunk = 4 * 10;
276         int scaleFactor = 10;
277         int* tris = (int*) calloc(chunk, sizeof(int));
278         int numCoords = 0;
279         int colourChange[numIterations];
280         int numChanges = 0;
281         FILE* triangles = fopen(filename, "rt");
282         if(triangles == NULL)
283         {
284             if(SHOW_MATH_ERRORS) printf("Error - Unable to open file containing
                    steps of traversed path.txt\n");
285         }
286         else
287         {
288             int i = 0;
289             char buffer[90];
290             while(fgets(buffer, 90, triangles) != NULL)
291             {
292                 if(buffer[0] != '\n')
293                 {
294                     int x = 0, y = 0;
295                     int numScanPts = sscanf(buffer, "%*s %d %d %*d", &x, &y);
296                     if(numScanPts == 2)
297                     {
298                         if(i%chunk == 0)
299                         {
300                             void* tmp = realloc(tris, (i + chunk) * sizeof(int));
301                             if(tmp != NULL)
302                                 tris = (int*) tmp;
303                             else
304                                 if(SHOW_MATH_ERRORS) printf("Error - Unable to
                                        reallocate memory for triangle coords\n");
305                         }
306
307                         if(x > moveSize || y > moveSize)
308                         {
309                             tris[i++] = x/scaleFactor;
310                             tris[i++] = y/scaleFactor;
311                             numCoords += 1;
312                         }
313                         else
314                         {
315                             tris[i++] = tris[i-2] + x;
316                             tris[i++] = tris[i-2] + y;
317                             numCoords += 1;
318                             if(i%chunk == 0)
319                             {
```

```
320                                 void* tmp = realloc(tris, (i + chunk) * sizeof(
                                        int));
321                                 if(tmp != NULL)
322                                     tris = (int*) tmp;
323                                 else
324                                     if(SHOW_MATH_ERRORS) printf("Error - Unable
                                            to reallocate memory for triangle coords\
                                            n");
325                             }
326                             tris[i++] = tris[i-2];
327                             tris[i++] = tris[i-2];
328                             numCoords += 1;
329                         }
330                     }
331                     else if(numScanPts == 0 && buffer[0] == 'T')
332                         colourChange[numChanges++] = numCoords;
333
334
335                     if(numChanges == numIterations)
336                     {
337                         ScaleTriangles(&tris, numCoords, 700, 700, 50, 50);//
                                scaled a lot of values to zero
338                         runDisplay(tris, numCoords, LINE_M, colourChange,
                                numChanges);
339                         i = 0, x = 0, y = 0;
340                         numChanges = 0;
341                         numCoords = 0;
342                     }
343                 }
344             }
345
346             fclose(triangles);
347         }
348
349         free(tris);
350         tris = NULL;
351     }
352
353     void append(char* s, char c)
354     {
355         int len = strlen(s);
356         s[len] = c;
357         s[len+1] = '\0';
358     }
359
360     int compare_ints (const void *a, const void *b)
361     {
```

```
362         const int *da = (const int *) a;
363         const int *db = (const int *) b;
364         return (*da > *db) - (*da < *db);
365     }
366
367     void calcStats(int* values, int numValues, int* med, double* mean, double*
            dev)
368     {
369         double sum = 0;
370         double sumOfSquares = 0;
371
372
373         qsort(values, numValues, sizeof(int), compare_ints);
374
375         if(numValues % 2 == 0)
376             *med = (values[numValues/2] + values[numValues/2 - 1])/2;
377         else
378             *med = values[(numValues - 1)/2];
379
380         for(int i = 0; i < numValues; i++)
381             sum += values[i];
382
383         *mean = sum/numValues;
384
385         for(int i = 0; i < numValues; i++)
386             sumOfSquares +=(values[i] - *mean) * (values[i] - *mean);
387
388         *dev = sqrt(sumOfSquares/numValues);
389     }
390
391     void combineStats(int numIter, char* directoryPath)
392     {
393         char stepCountFilePath[CHAR_ARRAY_CHUNK];
394         char resultsFilePath[CHAR_ARRAY_CHUNK];
395         char outputFilePath[CHAR_ARRAY_CHUNK];
396
397         strcpy(stepCountFilePath, directoryPath);
398         strcpy(resultsFilePath, directoryPath);
399         strcpy(outputFilePath, directoryPath);
400         strcat(stepCountFilePath, "step_counts.txt");
401         strcat(resultsFilePath, "SimResults.txt");
402         strcat(outputFilePath, "trial_stats.txt");
403
404         FILE* stepCount = fopen(stepCountFilePath, "rt");
405         FILE* results = fopen(resultsFilePath, "rt");
406         FILE* output = fopen(outputFilePath, "wt");
407         if(stepCount == NULL || results == NULL || output == NULL)
```

```
408         {
409             if(SHOW_MATH_ERRORS) printf("Error - Unable to open necessary files\n
                    ");
410         }
411         else
412         {
413             int i = 0, numSteps;
414             bool start = true, end = false;
415             char stepBuffer[90], resultsBuffer[90], lineBuffer[90];
416             int numTotal = 0, numSucc = 0, numHaz = 0, numStation = 0, numAlt =
                    0;
417             int success[numIter], hazard[numIter], stationary[numIter], alternate
                    [numIter];
418             int succMed = 0, stationMed = 0, altMed = 0, hazMed = 0;
419             double succMean = 0, succDev = 0, stationMean = 0, stationDev = 0,
                    altMean = 0, altDev = 0, hazMean = 0, hazDev = 0;
420             while(fgets(stepBuffer, 90, stepCount) != NULL && fgets(resultsBuffer
                    , 90, results) != NULL)
421             {
422                 if(resultsBuffer[0] != '\n')
423                 {
424                     sscanf(stepBuffer, "%d", &numSteps);
425                     sscanf(resultsBuffer, "%*s %s", lineBuffer);
426                     if(lineBuffer[0] == 'b')
427                     {
428                         stationary[numStation++] = numSteps;
429                     }
430                     else if(lineBuffer[0] == 'w')
431                     {
432                         alternate[numAlt++] = numSteps;
433                     }
434                     else if(lineBuffer[0] == 'e')
435                     {
436                         hazard[numHaz++] = numSteps;
437                     }
438                     else if(lineBuffer[0] == 's')
439                     {
440                         success[numSucc++] = numSteps;
441                     }
442
443                     if(++numTotal%numIter == 0)
444                     {
445                         if(numSucc != 0)
446                             calcStats(success, numSucc, &succMed, &succMean, &
                                    succDev);
447                         if(numStation != 0)
```

```
448                             calcStats(stationary, numStation, &stationMed, &
                                    stationMean, &stationDev);
449                         if(numAlt != 0)
450                             calcStats(alternate, numAlt, &altMed, &altMean, &
                                    altDev);
451                         if(numHaz != 0)
452                             calcStats(hazard, numHaz, &hazMed, &hazMean, &hazDev)
                                    ;
453
454                         fprintf(output, "Success - \n");
455                         fprintf(output, "Reached Goal - Occurrences: %d \tMedian:
                                %d\tMean: %.3f\tS.D: %.3f\n", numSucc, succMed,
                                succMean, succDev);
456                         fprintf(output, "Failure - \n");
457                         fprintf(output, "Became stationary - Occurrences: %d \
                                tMedian: %d\tMean: %.3f\tS.D: %.3f\n", numStation,
                                stationMed, stationMean, stationDev);
458                         fprintf(output, "Stuck alternating - Occurrences: %d \
                                tMedian: %d\tMean: %.3f\tS.D: %.3f\n", numAlt, altMed
                                , altMean, altDev);
459                         fprintf(output, "Hit hazard - Occurrences: %d\tMedian: %
                                d\tMean: %.3f\tS.D: %.3f\n\n\n", numHaz, hazMed,
                                hazMean, hazDev);
460
461                         succMed = 0; stationMed = 0; altMed = 0; hazMed = 0;
462                         succMean = 0; succDev = 0; stationMean = 0; stationDev =
                                0;
463                         altMean = 0; altDev = 0; hazMean = 0; hazDev = 0;
464
465                         numSucc = 0; numStation = 0; numAlt = 0; numHaz = 0;
466                     }
467                 }
468             }
469
470             if(numSucc != 0)
471                 calcStats(success, numSucc, &succMed, &succMean, &succDev);
472             if(numStation != 0)
473                 calcStats(stationary, numStation, &stationMed, &stationMean, &
                        stationDev);
474             if(numAlt != 0)
475                 calcStats(alternate, numAlt, &altMed, &altMean, &altDev);
476             if(numHaz != 0)
477                 calcStats(hazard, numHaz, &hazMed, &hazMean, &hazDev);
478
479             fprintf(output, "Success - \n");
480             fprintf(output, "Reached Goal - Occurrences: %d \tMedian: %d\tMean: -
                    %.3f\tS.D: %.3f\n", numSucc, succMed, succMean, succDev);
```

```
481             fprintf(output, "Failure - \n");
482             fprintf(output, "Became stationary - Occurrences: %d \tMedian: %d\
                    tMean: %.3f\tS.D: %.3f\n", numStation, stationMed, stationMean,
                    stationDev);
483             fprintf(output, "Stuck alternating - Occurrences: %d \tMedian: %d\
                    tMean: %.3f\tS.D: %.3f\n", numAlt, altMed, altMean, altDev);
484             fprintf(output, "Hit hazard - Occurrences: %d \tMedian: %d\tMean: %.3
                    f\tS.D: %.3f\n", numHaz, hazMed, hazMean, hazDev);
485
486             fclose(stepCount);
487             fclose(results);
488             fclose(output);
489         }
490     }
491
492     void analyseStats(int argc, char* argv[])
493     {
494         int numIter = 255;
495         char directoryPath[CHAR_ARRAY_CHUNK];
496         char inputFilePath[CHAR_ARRAY_CHUNK];
497         char outputFilePath[CHAR_ARRAY_CHUNK];
498         sscanf(argv[0], "%s", directoryPath);
499         if(argc > 1)
500             sscanf(argv[1], "%d", &numIter);
501
502         strcpy(inputFilePath, directoryPath);
503         strcpy(outputFilePath, directoryPath);
504         strcat(inputFilePath, "Actual_Steps.txt");
505         strcat(outputFilePath, "step_counts.txt");
506
507         FILE* moves = fopen(inputFilePath, "rt");
508         FILE* output = fopen(outputFilePath, "wt");
509
510         if(moves == NULL || output == NULL)
511         {
512             if(SHOW_MATH_ERRORS) printf("Error - Unable to open file containing
                    steps of traversed path or file for output stats\n");
513         }
514         else
515         {
516             int i = 0, numSteps = -2;
517             bool start = true, end = false;
518             char buffer[90];
519             while(fgets(buffer, 90, moves) != NULL)
520             {
521                 if(buffer[0] != '\n' && buffer[0] != ':')
522                 {
```

```
523                     if(buffer[0] == 'R')
524                     {
525                         if(start)
526                         {
527                             start = false;
528                             end = true;
529                             numSteps = -2;//since first and last line won't be
                                    moves/steps
530                         }
531                         else if(end)
532                         {
533                             fprintf(output, "%d\n", numSteps + 1);//since line of
                                    final position won't have been encountered
534                             numSteps = -2;
535                         }
536                     }
537                     else if(buffer[0] == 'T')
538                     {
539                         start = true;
540                         end = false;
541                         fprintf(output, "%d\n", numSteps);
542                     }
543                     else
544                         numSteps++;
545                 }
546             }
547
548             if(end)
549                 fprintf(output, "%d\n", numSteps + 1);//since line of final
                        position won't have been encountered
550
551             fclose(moves);
552             fclose(output);
553
554             combineStats(numIter, directoryPath);
555         }
556     }
557
558     void displayPaths(int argc, char* argv[])
559     {
560         int numIter = 255;
561         int moveSize = 10;
562         char configFilePath[CHAR_ARRAY_CHUNK];
563         sscanf(argv[0], "%s", configFilePath);
564         if(argc > 1)
565             sscanf(argv[1], "%d", &numIter);
566         if(argc > 2)
```

```
567             sscanf(argv[2], "%d", &moveSize);
568
569         recreateTraversedPath(configFilePath, numIter, moveSize);
570     }
571
572     void followPath(int argc, char* argv[])
573     {
574         int numIter = 255;
575         int moveSize = 10;
576         char configFilePath[CHAR_ARRAY_CHUNK];
577         sscanf(argv[0], "%s", configFilePath);
578         if(argc > 1)
579             sscanf(argv[1], "%d", &numIter);
580         if(argc > 2)
581             sscanf(argv[2], "%d", &moveSize);
582
583 //      RunReplay(configFilePath, numIter, moveSize);
584 // Need to have create a SimEnv and set it up also, ie configFile
585     }
586
587     int main(int argc, char* argv[])
588     {
589         printf("%d %d %d %d %d %d\n", (int) sizeof(bool), (int) sizeof(int), (int
                ) sizeof(long), (int) sizeof(long long), (int) sizeof(Coord), (int)
                sizeof(Coord*));
590
591         char mode[8];
592         if(argc > 2)
593         {
594             sscanf(argv[1], "%s", mode);
595             argv++; argv++;
596             argc -= 2;
597
598             switch(mode[1])
599             {
600                 case 'r':
601                     runSim(argc, argv);
602                     break;
603                 case 'd':
604                     displayPaths(argc, argv);
605                     break;
606                 case 's':
607                     analyseStats(argc, argv);
608                     break;
609 //                  case 'f':
610 //                      followPath(argv[2]);
611 //                      break;
```

```
612             }
613         }
614         else
615             printf("No inputs received. Correct usage is giving the path of the
                    file listing the configurations and optionally the number of runs
                    to occur for each pair of configurations\n");
616
617         return 1;
618     }
619
620     /** Creates the simulation environment, imports settings & sets up simulation
            accordingly
621     */
622     SimulationEnvironment::SimulationEnvironment()
623     {
624         view = 0;
625         runSim = true;
626         runRobot = true;
627         allowInactiveScreen = false;
628         camera = (scene::ICameraSceneNode**) calloc(3, sizeof(scene::
                ICameraSceneNode*));
629         //Two user-controlled cameras (robot controlled, spectator controlled)
                and one fixed for top down overview
630         //could one the robot one be a CameraSceneNode rather than
                CameraSceneNodeFPS as doesn't need user control, just be able to move
631     }
632
633     /** Creates the simulation environment, imports settings & sets up simulation
            accordingly
634     */
635     SimulationEnvironment::SimulationEnvironment(char* configFilePath, char*
            obstaclesFilePath)
636     {
637         view = 0;
638         runSim = true;
639         runRobot = true;
640         allowInactiveScreen = false;
641         camera = (scene::ICameraSceneNode**) calloc(3, sizeof(scene::
                ICameraSceneNode*));
642         //Two user-controlled cameras (robot controlled, spectator controlled)
                and one fixed for top down overview
643         //could one the robot one be a CameraSceneNode rather than
                CameraSceneNodeFPS as doesn't need user control, just be able to move
644
645 //      SetupEnvironment(configFilePath, obstaclesFilePath);
646     }
647
```

```
648     /** Is fine if ExitSim has been called as objects set to NULL and delete/free
            on NULL object are valid/ok
649     */
650     SimulationEnvironment::~SimulationEnvironment()
651     {
652         free(sConfig.heightMap);
653         free(sConfig.textureOverlay);
654         free(sConfig.closeTextureOverlay);
655         free(camera);
656         camera = NULL;
657         delete robot;
658         robot = NULL;
659     }
660
661     /** Opens a config file and parses the contents of it
662     */
663     void SimulationEnvironment::ImportSettingsConfig(char* configFilePath)
664     {
665         FILE* configFile;
666         configFile = fopen(configFilePath, "rt");
667         if (configFile == NULL)
668         {
669             if(SHOW_SIM_ERRORS) printf("Error - Unable to open settings config
                    file\n");
670             exit(1); // terminate with error
671         }
672
673         sConfig.algorithm = 0;
674         sConfig.heightMap = NULL, sConfig.textureOverlay = NULL, sConfig.
                closeTextureOverlay = NULL;
675         sConfig.startPosition = core::vector3df(0,0,0), sConfig.goalPosition =
                core::vector3df(0,0,0);
676
677         char buffer[CHAR_ARRAY_CHUNK];
678         char field[CHAR_ARRAY_CHUNK];
679         while(fgets(buffer, CHAR_ARRAY_CHUNK, configFile) != NULL)
680         {
681             if(buffer[0] != '\n')
682             {
683                 sscanf(buffer, "%s", field);
684                 switch(field[0])
685                 {
686                     case 'A':
687                     case 'a':
688                     {
689                         int alg;
690                         sscanf (buffer, "%*s %d", &alg);
```

```
691                         sConfig.algorithm = alg;
692                         break;
693                     }
694                     case 'S':
695                     case 's':
696                     {
697                         float x,y,z;
698                         sscanf (buffer, "%*s %f %f %f", &x,&y,&z);
699                         sConfig.startPosition = core::vector3df(x,z,y);
700                         printf("Values are %f %f %f %f %f %f\n", x,y,z, sConfig.
                                startPosition.X, sConfig.startPosition.Y, sConfig.
                                startPosition.Z );
701                         break;
702                     }
703                     case 'G':
704                     case 'g':
705                     {
706                         float x,y,z;
707                         sscanf (buffer, "%*s %f %f %f", &x,&y,&z);
708                         sConfig.goalPosition = core::vector3df(x,z,y);
709                         break;
710                     }
711                     case 'C':
712                     case 'c':
713                     {
714                         int camNum;
715                         float xPos, yPos, zPos, xTgt, yTgt, zTgt, farValue;
716                         sscanf (buffer, "%*s %d %f %f %f %f %f %f %f", &camNum, &
                                xPos,&yPos,&zPos, &xTgt, &yTgt, &zTgt, &farValue);
717                         sConfig.cameraPositions[camNum] = core::vector3df(xPos,
                                zPos,yPos);
718                         sConfig.cameraTargets[camNum] = core::vector3df(xTgt,
                                zTgt, yTgt);
719                         sConfig.cameraFarValue[camNum] = farValue;
720                         break;
721                     }
722                     case 'H':
723                     case 'h':
724                     {
725                         char temp[CHAR_ARRAY_CHUNK];
726                         sscanf (buffer, "%*s %s", temp);
727
728                         int i = 0;
729                         for(i = 0; temp[i] != '\0'; i++);
730
731                         if(i == 0 || i >= CHAR_ARRAY_CHUNK)
732                             ExitSim();
```

```
733
734                         sConfig.heightMap = (char*) calloc(i+1, sizeof(char));///
                                add a free() somewhere
735                         for(int j = 0; j <= i; j++)
736                             sConfig.heightMap[j] = temp[j];
737                         break;
738                     }
739                     case 'T':
740                     case 't':
741                     {
742                         int textureNum;
743                         char temp[CHAR_ARRAY_CHUNK];
744                         sscanf (buffer, "%*s %d %s", &textureNum, temp);
745
746                         int i = 0;
747                         for(i = 0; temp[i] != '\0'; i++);
748
749                         if(i >= CHAR_ARRAY_CHUNK)
750                             ExitSim();
751
752                         if(textureNum == 0)
753                         {
754                             sConfig.textureOverlay = (char*) calloc(i+1, sizeof(
                                    char));///add a free() somewhere
755                             for(int j = 0; j <= i; j++)
756                                 sConfig.textureOverlay[j] = temp[j];
757                         }
758                         else
759                         {
760                             sConfig.closeTextureOverlay = (char*) calloc(i+1,
                                    sizeof(char));///add a free() somewhere
761                             for(int j = 0; j <= i; j++)
762                                 sConfig.closeTextureOverlay[j] = temp[j];
763                         }
764                         break;
765                     }
766                     case '\n':
767                         break;
768                     default:
769                     {
770                         if(SHOW_SIM_ERRORS) printf("Error - Config file contains
                                unknown field\n");
771                         break;
772                     }
773                 }
774             }
775         }
```

```
776         fclose(configFile);
777     }
778
779     void SimulationEnvironment::ImportObstaclesConfig(char* obstaclesFilePath,
            ObstacleConfig** oConfig, int* numObstacles)
780     {
781         FILE* file;
782         file = fopen(obstaclesFilePath, "rt");
783         if (file == NULL)
784         {
785             FILE* out = fopen("Errors.txt", "at");
786             fprintf(out, "file path = %s\n", obstaclesFilePath);
787             if(SHOW_SIM_ERRORS) printf("Error - Unable to open obstacles config
                    file\n");
788             exit(1); // terminate with error
789         }
790
791         char buffer[CHAR_ARRAY_CHUNK];
792         char field[CHAR_ARRAY_CHUNK];
793         while(fgets(buffer, CHAR_ARRAY_CHUNK, file) != NULL)
794         {
795             if(buffer[0] != '\n')
796             {
797                 sscanf(buffer, "%s", field);
798                 switch(field[0])
799                 {
800
801                     case '<':
802                     {
803                         if(field[1] == '/')
804                             (*numObstacles)++;
805                         else if(*numObstacles != 0 && (*numObstacles)%LIST_CHUNK
                                == 0 )
806                         {
807                             void* tmp = realloc(*oConfig, (*numObstacles +
                                    LIST_CHUNK)*sizeof(ObstacleConfig));
808                             if(tmp != NULL)
809                                 *oConfig = (ObstacleConfig*) tmp;
810                         }
811                         break;
812                     }
813                     case 'P':
814                     case 'p':
815                     {
816                         float x,y,z;
817                         sscanf (buffer, "%*s %f %f %f", &x,&y,&z);
```

```
818                     (*oConfig)[*numObstacles].position = core::vector3df(x,z,
                            y);
819                     break;
820                 }
821                 case 'R':
822                 case 'r':
823                 {
824                     float x,y,z;
825                     sscanf (buffer, "%*s %f %f %f", &x,&y,&z);
826                     (*oConfig)[*numObstacles].rotation = core::vector3df(x,z,
                            y);
827                     break;
828                 }
829                 case 'S':
830                 case 's':
831                 {
832                     float x,y,z;
833                     sscanf (buffer, "%*s %f %f %f", &x,&y,&z);
834                     (*oConfig)[*numObstacles].scale = core::vector3df(x,z,y);
835                     break;
836                 }
837                 case 'M':
838                 case 'm':
839                 {
840                     char temp[CHAR_ARRAY_CHUNK];
841                     sscanf (buffer, "%*s %s", temp);
842 
843                     int i = 0;
844                     for(i = 0; temp[i] != '\0'; i++);
845 
846                     if(i == 0 || i >= CHAR_ARRAY_CHUNK)
847                         ExitSim();
848 
849                     (*oConfig)[*numObstacles].mesh = (char*) calloc(i+1,
                            sizeof(char));///add a free() somewhere
850                     for(int j = 0; j <= i; j++)
851                         (*oConfig)[*numObstacles].mesh[j] = temp[j];
852 
853                     break;
854                 }
855                 case 'T':
856                 case 't':
857                 {
858                     char temp[CHAR_ARRAY_CHUNK];
859                     sscanf (buffer, "%*s %s", temp);
860 
861                     int i = 0;
```

```
862                         for(i = 0; temp[i] != '\0'; i++);
863
864                         if(i >= CHAR_ARRAY_CHUNK)
865                             ExitSim();
866
867
868                         (*oConfig)[*numObstacles].textureOverlay = (char*) calloc
                                (i+1, sizeof(char));///add a free() somewhere
869                         for(int j = 0; j <= i; j++)
870                             (*oConfig)[*numObstacles].textureOverlay[j] = temp[j
                                    ];
871                         break;
872                     }
873                     case '\n':
874                         break;
875                     default:
876                     {
877                         if(SHOW_SIM_ERRORS) printf("Error - Config file contains
                                unknown field\n");
878                         break;
879                     }
880                 }
881             }
882         }
883         fclose(file);
884     }
885
886     int SimulationEnvironment::SetupEnvironment(char* configFilePath, char*
            obstaclesFilePath)
887     {
888         if(DEBUG)
889             driverType = video::EDT_OPENGL;
890         else
891         {
892             printf("Please select the driver you want for this example:\n"\
893                 "(a) Direct3D 9.0c\n(b) Direct3D 8.1\n(c) OpenGL 1.5\n"\
894                 "(d) Software Renderer\n(e) Burning's Software Renderer\n"\
895                 "(f) NullDevice\n(otherKey) exit\n\n");
896             char i;
897             std::cin >> i;
898             switch(i)
899             {
900                 case 'a': driverType = video::EDT_DIRECT3D9;break;
901                 case 'b': driverType = video::EDT_DIRECT3D8;break;
902                 case 'c': driverType = video::EDT_OPENGL;   break;
903                 case 'd': driverType = video::EDT_SOFTWARE; break;
904                 case 'e': driverType = video::EDT_BURNINGSVIDEO;break;
```

```
905                 case 'f': driverType = video::EDT_NULL;     break;
906                 default: return 1;
907             }
908         }
909
910         core::dimension2d<s32> dim = core::dimension2d<s32>(WINDOW_WIDTH,
                WINDOW_HEIGHT);
911         device = createDevice(driverType, dim);
912         if (device == 0)
913             return 1; // could not create selected driver.
914
915         driver = device->getVideoDriver();
916         driver->setTextureCreationFlag(video::ETCF_ALWAYS_32_BIT, true);
917         smgr = device->getSceneManager();
918
919         ImportSettingsConfig(configFilePath);
920         camera[0] = smgr->addCameraSceneNodeFPS(0,100.0f,1.2f);
921         camera[0]->setPosition(sConfig.cameraPositions[0]);
922         camera[0]->setTarget(sConfig.cameraTargets[0]);
923         camera[0]->setFarValue(sConfig.cameraFarValue[0]);
924         camera[1] = smgr->addCameraSceneNodeFPS(0,100.f,1.2f);
925         camera[1]->setPosition(sConfig.cameraPositions[1]);
926         camera[1]->setTarget(sConfig.cameraTargets[1]);
927         camera[1]->setFarValue(42000.0f);
928         camera[2] = smgr->addCameraSceneNode(0,sConfig.cameraPositions[2],sConfig
                .cameraTargets[2]);
929         camera[2]->setFarValue(42000.0f);
930
931         device->getCursorControl()->setVisible(false);
932
933         metaSelector = smgr->createMetaTriangleSelector();
934         numAnim = 0;
935         anim = (scene::ISceneNodeAnimatorCollisionResponse**) calloc(LIST_CHUNK,
                sizeof(scene::ISceneNodeAnimatorCollisionResponse*));
936
937         terrain = smgr->addTerrainSceneNode(sConfig.heightMap,0,-1,core::
                vector3df(0.f, 0.f, 0.f), core::vector3df(0.f, 0.f, 0.f), core::
                vector3df(40.f, 4.4f, 40.f), video::SColor ( 255, 255, 255, 255 ), 5,
                 scene::ETPS_17, 4);
938         if(terrain != NULL)
939         {
940             terrain->setMaterialFlag(video::EMF_LIGHTING, false);
941             terrain->setMaterialTexture(0, driver->getTexture(sConfig.
                    textureOverlay));
942             terrain->setMaterialTexture(1, driver->getTexture(sConfig.
                    closeTextureOverlay));
943             terrain->setMaterialType(video::EMT_DETAIL_MAP);
```

```
944             terrain->scaleTexture(1.0f, 20.0f);
945             scene::ITriangleSelector* selector = smgr->
                    createTerrainTriangleSelector(terrain, 0);
946             terrain->setTriangleSelector(selector);
947             metaSelector->addTriangleSelector(selector);
948             selector->drop();
949         }
950         else
951             return 1;
952
953         int startLine[3] = {sConfig.startPosition.X, sConfig.startPosition.Y,
                sConfig.startPosition.Z};
954         int endLine[3] = {sConfig.startPosition.X, sConfig.startPosition.Y - 1,
                sConfig.startPosition.Z};
955         float dist = CheckForCollision(startLine, endLine);
956
957         int watchDog = 0, incrementalOffset = 0;
958         while(dist == -1 && watchDog++ < 100)//seems to usually be due to
                collision point being so close to current height value
959         {
960             incrementalOffset += 100;
961             startLine[1] = sConfig.startPosition.Y + incrementalOffset;
962             endLine[1] = startLine[1] - 1;
963             dist = CheckForCollision(startLine, endLine);
964         }
965         if(dist == -1)
966             incrementalOffset = 0;
967
968         sConfig.startPosition.Y = sConfig.startPosition.Y + incrementalOffset -
                dist;
969         core::vector3df robotNodePosition = sConfig.startPosition;
970         robotNodePosition.Y += ROBOTNODE_VERTBOOST;
971         sConfig.cameraPositions[0].Y = robotNodePosition.Y + CAMERA_VERTBOOST;
972         sConfig.cameraTargets[0].Y = sConfig.cameraPositions[0].Y;
973         camera[0]->setPosition(sConfig.cameraPositions[0]);
974         camera[0]->setTarget(sConfig.cameraTargets[0]);
975
976         scene::IAnimatedMesh* carMesh = smgr->getMesh("../media/porsche.obj");
977         robotNode = smgr->addAnimatedMeshSceneNode(carMesh);
978         if(robotNode != NULL)
979         {
980             robotNode->setPosition(sConfig.startPosition);
981             robotNode->setMaterialFlag(video::EMF_LIGHTING, false);
982             robotNode->setMaterialType(video::EMT_DETAIL_MAP);
983         }
984         else
985             return 1;
```

```
986
987         scene::IAnimatedMesh* goalFlag = smgr->getMesh("../media/tall_box.3ds");
988         scene::ISceneNode* tNode = smgr->addAnimatedMeshSceneNode(goalFlag);
989         if(tNode != NULL)
990         {
991             startLine[0] = sConfig.goalPosition.X; startLine[1] = sConfig.
                goalPosition.Y; startLine[2] = sConfig.goalPosition.Z;
992             endLine[0] = startLine[0]; endLine[1] = startLine[1] - 1; endLine[2]
                = startLine[2];
993             dist = CheckForCollision(startLine, endLine);
994
995             watchDog = 0;
996             incrementalOffset = 0;
997             while(dist == -1 && watchDog++ < 100)//seems to usually be due to
                collision point being so close to current height value
998             {
999                 incrementalOffset += 100;
1000                startLine[1] = sConfig.goalPosition.Y + incrementalOffset;
1001                endLine[1] = startLine[1] - 1;
1002                dist = CheckForCollision(startLine, endLine);
1003            }
1004            if(dist == -1)
1005                incrementalOffset = 0;
1006
1007            sConfig.goalPosition.Y = sConfig.goalPosition.Y + incrementalOffset -
                 dist;
1008            tNode->setPosition(sConfig.goalPosition);
1009            tNode->setMaterialFlag(video::EMF_LIGHTING, false);
1010            tNode->setMaterialType(video::EMT_TRANSPARENT_ADD_COLOR);
1011            tNode->setMaterialTexture(0, driver->getTexture("../media/wall.bmp"))
                ;
1012            tNode->setScale(core::vector3df(20,75,20));
1013
1014        }
1015
1016        int numObstacles = 0;
1017        ObstacleConfig* oConfig = (ObstacleConfig*) calloc(LIST_CHUNK, sizeof(
            ObstacleConfig));
1018        ImportObstaclesConfig(obstaclesFilePath, &oConfig, &numObstacles);
1019        AddObstacles(oConfig, numObstacles);
1020        for(int i = 0; i < numObstacles; i++)
1021        {
1022            free(oConfig[i].mesh);
1023            free(oConfig[i].textureOverlay);
1024        }
1025        free(oConfig);
1026
```

```
1027        for(int i = 0; i < numAnim; i++)
1028            anim[i]->drop();
1029        metaSelector->drop();
1030
1031        // create skybox & skydome
1032        driver->setTextureCreationFlag(video::ETCF_CREATE_MIP_MAPS, false);
1033
1034        skybox = smgr->addSkyBoxSceneNode(
1035            driver->getTexture("../media/irrlicht2_up.jpg"),
1036            driver->getTexture("../media/irrlicht2_dn.jpg"),
1037            driver->getTexture("../media/irrlicht2_lf.jpg"),
1038            driver->getTexture("../media/irrlicht2_rt.jpg"),
1039            driver->getTexture("../media/irrlicht2_ft.jpg"),
1040            driver->getTexture("../media/irrlicht2_bk.jpg"));
1041            //reverse lf and rt order if switch between irr v1.5 differs to v1.3
1042
1043        skydome = smgr->addSkyDomeSceneNode(driver->getTexture("../media/skydome.
                jpg"),16,8,0.95f,2.0f);
1044        skydome->setRotation(core::vector3df(0,180,0));
1045
1046        skybox->setVisible(true);
1047        skydome->setVisible(false);
1048
1049        driver->setTextureCreationFlag(video::ETCF_CREATE_MIP_MAPS, true);
1050        smgr->setActiveCamera(camera[0]);
1051
1052        SetupGui();
1053        SetupRobot();
1054
1055        return 0;
1056    }
1057
1058    /** Adds obstacle objects to terrain
1059        Will need to load info from a config file
1060        Also will have to realloc anim[] if gets too big
1061    */
1062    void SimulationEnvironment::AddObstacles(ObstacleConfig* obstacles, int
            numObstacles)
1063    {
1064        if(numObstacles > 0)
1065            for(int i = 0; i < numObstacles; i++)
1066            {
1067                scene::ISceneNode *boxNode = 0;
1068                scene::IAnimatedMesh* boxMesh = smgr->getMesh(obstacles[i].mesh);
1069                boxNode = smgr->addAnimatedMeshSceneNode(boxMesh);
1070                if(boxNode != NULL)
1071                {
```

```
1072                boxNode->setMaterialFlag(video::EMF_LIGHTING, false);
1073                boxNode->setMaterialType(video::EMT_DETAIL_MAP);
1074                boxNode->setMaterialTexture(0, driver->getTexture(obstacles[i
                        ].textureOverlay));
1075
1076                boxNode->setScale(obstacles[i].scale);
1077                boxNode->setRotation(obstacles[i].rotation);
1078                core::aabbox3df bbox = boxNode->getBoundingBox();
1079
1080                int startLine[3] = {obstacles[i].position.X, obstacles[i].
                        position.Y, obstacles[i].position.Z};
1081                int endLine[3] = {obstacles[i].position.X, obstacles[i].
                        position.Y - 1, obstacles[i].position.Z};
1082                float dist = CheckForCollision(startLine, endLine);
1083                int watchDog = 0, incrementalOffset = 0;
1084                while(dist == -1 && watchDog++ < 100)//seems to usually be
                        due to collision point being so close to current height
                        value
1085                {
1086                    incrementalOffset += 100;
1087                    startLine[1] = obstacles[i].position.Y +
                            incrementalOffset;
1088                    endLine[1] = startLine[1] - 1;
1089                    dist = CheckForCollision(startLine, endLine);
1090                }
1091                if(dist == -1)
1092                {
1093                    printf("Error - Correct height for obstacle could not be
                            calcuated\n");
1094                    incrementalOffset = 0;
1095                }
1096
1097                obstacles[i].position.Y = obstacles[i].position.Y +
                        incrementalOffset - dist;
1098                boxNode->setPosition(obstacles[i].position);
1099
1100                scene::ITriangleSelector* selector = smgr->
                        createOctTreeTriangleSelector(boxMesh->getMesh(0),
                        boxNode, 128);
1101                boxNode->setTriangleSelector(selector);
1102                metaSelector->addTriangleSelector(selector);
1103                selector->drop();
1104
1105                anim[numAnim] = smgr->createCollisionResponseAnimator(
1106                    metaSelector, boxNode,
1107                    bbox.MaxEdge - bbox.getCenter(),
1108                    core::vector3df(0,-10.0f,0),
```

```
1109                        core::vector3df(0,0,0));
1110                    boxNode->addAnimator(anim[numAnim++]);
1111
1112                    if(numAnim%LIST_CHUNK == 0)
1113                    {
1114                        void* tmp = realloc(anim, (numAnim + LIST_CHUNK)*sizeof(
                                scene::ISceneNodeAnimatorCollisionResponse*));
1115                        if(tmp != NULL)
1116                            anim = (scene::ISceneNodeAnimatorCollisionResponse**)
                                    tmp;
1117                    }
1118                }
1119            }
1120        else
1121        {
1122            int startLine[3], endLine[3];
1123            startLine[0] = 4000; startLine[1] = 1200; startLine[2] = 7800;
1124            endLine[0] = startLine[0]; endLine[1] = startLine[1] - 1; endLine[2]
                = startLine[2];
1125            float dist = CheckForCollision(startLine, endLine);
1126            startLine[1] -= dist;
1127            scene::ISceneNode *boxNode1 = 0, *boxNode2 = 0;
1128            scene::IAnimatedMesh* boxMesh = smgr->getMesh("../media/tall_box.3ds"
                );
1129            boxNode1 = smgr->addAnimatedMeshSceneNode(boxMesh);
1130            if(boxNode1 != NULL)
1131            {
1132                boxNode1->setMaterialFlag(video::EMF_LIGHTING, false);
1133                boxNode1->setMaterialType(video::EMT_DETAIL_MAP);
1134                boxNode1->setMaterialTexture(0, driver->getTexture("../media/wall
                    .bmp"));
1135
1136                boxNode1->setPosition(core::vector3df(startLine[0], startLine[1],
                     startLine[2]));
1137                boxNode1->setRotation(core::vector3df(0,0,0));
1138                boxNode1->setScale(core::vector3df(100,100,100));
1139
1140                scene::ITriangleSelector* selector = smgr->
                    createOctTreeTriangleSelector(boxMesh->getMesh(0), boxNode1,
                    128);
1141                boxNode1->setTriangleSelector(selector);
1142                metaSelector->addTriangleSelector(selector);
1143                selector->drop();
1144            }
1145
1146            anim[numAnim] = smgr->createCollisionResponseAnimator(
1147                metaSelector, boxNode1,
```

```
1148                boxNode1->getBoundingBox().MaxEdge - boxNode1->getBoundingBox().
                        getCenter(),
1149                core::vector3df(0,-10.0f,0),
1150                core::vector3df(0,0,0));
1151            boxNode1->addAnimator(anim[numAnim++]);
1152        }
1153    }
1154
1155    void SimulationEnvironment::RunEnvironment()
1156    {
1157        int lastFPS = -1;
1158        uint numLoops = 0;
1159        bool goalReached = false;
1160        MyEventReceiver receiver(terrain, smgr, env, this);
1161        device->setEventReceiver(&receiver);
1162
1163        Coord rPos = robot->GetPosition();
1164        int failDistance = CalculateDirectDistance(rPos, robot->GetGoal(), true)
                /100 * 2;
1165        robot->RecordData("Desired_Steps.txt", rPos.xVal, rPos.yVal, rPos.zVal);
1166        robot->RecordData("Actual_Steps.txt", rPos.xVal, rPos.yVal, rPos.zVal);
1167        robot->RecordData("Way_Points.txt", rPos.xVal, rPos.yVal, rPos.zVal);
1168        WriteTriToFile<Face_handle>("hazards.txt", "wt", NULL, 0, true);
1169
1170        while(device->run() && !goalReached)
1171        if (runSim && (allowInactiveScreen || device->isWindowActive()))
1172        {
1173            driver->beginScene(true, true, 0 );
1174            if(smgr == NULL)
1175                printf("********* Run() - SMGR == NULL *********\n");
1176
1177
1178            ///Split screen - try to use to have portion displaying info
1179            driver->setViewPort(core::rect<s32>(0,0,WINDOW_WIDTH,VIEW_HEIGHT));
1180            smgr->drawAll();
1181
1182            driver->setViewPort(core::rect<s32>(0,0,WINDOW_WIDTH,WINDOW_HEIGHT));
1183            UpdateGui();
1184            env->drawAll();
1185
1186            driver->endScene();
1187
1188            // display frames per second in window title
1189            int fps = driver->getFPS();
1190            if (lastFPS != fps)
1191            {
```

```
1192                core::stringw str = L"Path Planning Simulator on Irrlicht Engine
                        [";
1193                str += driver->getName();
1194                str += "] FPS:";
1195                str += fps;
1196                str += " Height: ";
1197                str += terrain->getHeight(camera[view]->getAbsolutePosition().X,
1198                    camera[view]->getAbsolutePosition().Z);
1199
1200                device->setWindowCaption(str.c_str());
1201                lastFPS = fps;
1202            }
1203
1204            if(robot != NULL && runRobot)
1205            {
1206                if(numLoops > failDistance)
1207                {
1208                    device->closeDevice();
1209                }
1210                else if(numLoops > 0 && numLoops%10 == 0)
1211                {
1212                    PanView();
1213                    robot->terrain->CullHeldNodes();
1214                }
1215
1216                if(numLoops > 10)//perhaps have so moves a fraction each loop and
                        on every Nth one redoes gather data and plan path
1217                {
1218                    goalReached = UpdateRobot();
1219                }
1220
1221                numLoops++;
1222            }
1223        }
1224
1225        CloseEnvironment();
1226    }
1227
1228    void SimulationEnvironment::RunReplay(char* inputFilePath, int numIter, int
        moveSize)
1229    {
1230        int lastFPS = -1;
1231        uint numLoops = 0;
1232        bool goalReached = false;
1233        char buffer[90];
1234        FILE* path = fopen(inputFilePath, "rt");
1235        if(path == NULL)
```

```
1236        {
1237            if(SHOW_MATH_ERRORS) printf("Error - Unable to open file containing
                    steps of traversed path.txt\n");
1238            return;
1239        }
1240
1241        MyEventReceiver receiver(terrain, smgr, env, this);
1242        device->setEventReceiver(&receiver);
1243        while(device->run() && !goalReached)
1244        if (runSim && (allowInactiveScreen || device->isWindowActive()))
1245        {
1246            driver->beginScene(true, true, 0 );
1247            if(smgr == NULL)
1248                printf("********* Run() - SMGR == NULL *********\n");
1249
1250            ///Split screen - try to use to have portion displaying info
1251            driver->setViewPort(core::rect<s32>(0,0,WINDOW_WIDTH,VIEW_HEIGHT));
1252            smgr->drawAll();
1253
1254            driver->setViewPort(core::rect<s32>(0,0,WINDOW_WIDTH,WINDOW_HEIGHT));
1255            UpdateGui();
1256            env->drawAll();
1257            driver->endScene();
1258
1259            // display frames per second in window title
1260            int fps = driver->getFPS();
1261            if (lastFPS != fps)
1262            {
1263                core::stringw str = L"Path Planning Simulator on Irrlicht Engine
                        [";
1264                str += driver->getName();
1265                str += "] FPS:";
1266                str += fps;
1267                str += " Height: ";
1268                str += terrain->getHeight(camera[view]->getAbsolutePosition().X,
1269                    camera[view]->getAbsolutePosition().Z);
1270                device->setWindowCaption(str.c_str());
1271                lastFPS = fps;
1272            }
1273
1274
1275            if(fgets(buffer, 90, path) != NULL)
1276            {
1277                if(buffer[0] != '\n')
1278                {
1279                    core::vector3df step, newPos;
1280                    int x = 0, y = 0, z = 0;
```

```
1281                    int numScanPts = sscanf(buffer, "%*s %d %d %d", &x, &y, &z);
1282                    if(numScanPts == 3)
1283                    {
1284                        if(x > moveSize || y > moveSize)
1285                        {
1286                            step = core::vector3df(0, 0, 0);
1287                            newPos = core::vector3df(x/moveSize, z/moveSize, y/
                                moveSize);
1288                        }
1289                        else
1290                        {
1291                            step = core::vector3df(x, z, y);
1292                            printf("Moved %d %d %d\n", (int) step.X, (int) step.Z
                                , (int) step.Y);
1293                            newPos = robotNode->getPosition() + step;
1294                        }
1295
1296                        robotNode->setPosition(newPos);
1297                        core::vector3df newCamPos = newPos + core::vector3df(0,
                            CAMERA_VERTBOOST,0);
1298                        core::vector3df newCamTarget = newPos + step + core::
                            vector3df(0,CAMERA_VERTBOOST,0);
1299                        newCamTarget.Y = newCamPos.Y; //To have camera looking
                            horizontally, ie rather than skyward when climbing a
                            rise
1300                        camera[0]->setPosition(newCamPos);
1301                        camera[0]->setTarget(newCamTarget);
1302                    }
1303                }
1304            }
1305            else
1306            {
1307                goalReached = true;
1308                device->closeDevice();
1309            }
1310        }
1311
1312        CloseEnvironment();
1313    }
1314
1315    void SimulationEnvironment::SetupGui()
1316    {
1317        env = device->getGUIEnvironment();
1318        env->getSkin()->setFont(env->getFont("../media/fontlucida.png"));
1319
1320        env->addStaticText(L"Mode: ", core::rect<s32>(GAP, VIEW_HEIGHT + GAP, GAP
                + 45, VIEW_HEIGHT + GAP + LINE_HEIGHT), true, true, 0, -1, true);
```

```
1321        env->addStaticText(L"No Mode Selected", core::rect<s32>(GAP + 45 ,
                VIEW_HEIGHT + GAP, GAP + COLUMN_WIDTH, VIEW_HEIGHT + GAP +
                LINE_HEIGHT), true, true, env->getRootGUIElement(), GUI_ID_MODE, true
                );
1322        env->addStaticText(L"Press:\n '1' to see robot's view\n '2' for user
                controlled camera\n '3' for bird's eye view\n 'C' to close the
                simulation", core::rect<s32>(GAP, VIEW_HEIGHT + GAP + LINE_HEIGHT,
                GAP + COLUMN_WIDTH, WINDOW_HEIGHT - GAP), true, true, 0, GUI_ID_CTRLS
                , true);
1323    }
1324
1325    void SimulationEnvironment::SetupRobot()
1326    {
1327        Coord position, gps, dest;
1328        position = FillCoord(sConfig.startPosition.X,sConfig.startPosition.Z,
                sConfig.startPosition.Y, SCALE_FACTOR);
1329        gps = FillCoord(0, 0, 0, SCALE_FACTOR);
1330        dest = FillCoord(sConfig.goalPosition.X, sConfig.goalPosition.Z, sConfig.
                goalPosition.Y, SCALE_FACTOR);
1331
1332        robot = new Robot(gps, position, dest, this);
1333
1334        //Get current pos, plus four other starting ones
1335        Coord someStartCoords[5];
1336        someStartCoords[0] = position;
1337
1338        // 15 & 25 as those are half the robot width and length. 2000 so as to
                ensure is above height of terrain at that point
1339        int startLine[3] = {sConfig.startPosition.X - 15, sConfig.startPosition.Y
                + 2000, sConfig.startPosition.Z - 25};
1340        int endLine[3] = {startLine[0], startLine[1] - 1, startLine[2]};
1341        float dist = CheckForCollision(startLine, endLine);
1342        someStartCoords[1] = FillCoord(startLine[0], startLine[2], startLine[1] -
                dist, SCALE_FACTOR);
1343
1344        startLine[0] = sConfig.startPosition.X + 15; startLine[1] = sConfig.
                startPosition.Y + 2000; startLine[2] = sConfig.startPosition.Z - 25;
1345        endLine[0] = startLine[0]; endLine[1] = startLine[1] - 1; endLine[2] =
                startLine[2];
1346        dist = CheckForCollision(startLine, endLine);
1347        someStartCoords[2] = FillCoord(startLine[0], startLine[2], startLine[1] -
                dist, SCALE_FACTOR);
1348
1349        startLine[0] = sConfig.startPosition.X - 15; startLine[1] = sConfig.
                startPosition.Y + 2000; startLine[2] = sConfig.startPosition.Z + 25;
1350        endLine[0] = startLine[0]; endLine[1] = startLine[1] - 1; endLine[2] =
                startLine[2];
```

```
1351        dist = CheckForCollision(startLine, endLine);
1352        someStartCoords[3] = FillCoord(startLine[0], startLine[2], startLine[1] -
                dist, SCALE_FACTOR);
1353
1354        startLine[0] = sConfig.startPosition.X + 15; startLine[1] = sConfig.
                startPosition.Y + 2000; startLine[2] = sConfig.startPosition.Z + 25;
1355        endLine[0] = startLine[0]; endLine[1] = startLine[1] - 1; endLine[2] =
                startLine[2];
1356        dist = CheckForCollision(startLine, endLine);
1357        someStartCoords[4] = FillCoord(startLine[0],startLine[2], startLine[1] -
                dist, SCALE_FACTOR);
1358
1359        robot->GatherData(someStartCoords, 5);
1360    }
1361
1362    void SimulationEnvironment::UpdateGui()
1363    {
1364    }
1365
1366    bool SimulationEnvironment::UpdateRobot()
1367    {
1368        bool goalReached = false;
1369        robot->GatherData(this, INPUT_TYPE);
1370        robot->PlanPath();
1371        Coord shift = robot->MoveDirection();
1372
1373        if(shift.x() == 0 && shift.y() == 0 && shift.z() == 0)//this one if no
                path found, the step = 0 when have reached end ie z might be off?
1374        {
1375            if(robot->NaviDone())
1376            {
1377                goalReached = true;
1378                EndRun(goalReached, NULL);
1379            }
1380            else if(robot->consecutiveTimesStationary++ == 10)
1381            {
1382                EndRun(goalReached, "Robot␣became␣stationary␣as␣could␣not␣
                        determine␣path␣to␣goal.␣");
1383            }
1384            else
1385            {
1386                Coord dirVect = robot->GetDirection();
1387                if(dirVect == robot->GetGoal() - robot->GetPosition())
1388                {
1389                    //calc norm vect
1390                    dirVect = NormaliseVector<Coord>(dirVect, MOVE_INCREMENTS_MM/
                            SCALE_FACTOR, false);
```

```
1391                    Coord normVect = CrossProduct(dirVect, dirVect + Coord(0,0,
                            MOVE_INCREMENTS_MM/SCALE_FACTOR));
1392                    normVect = NormaliseVector<Coord>(normVect,
                            MOVE_INCREMENTS_MM/SCALE_FACTOR, false);
1393                    dirVect = dirVect + normVect;//results in 45 degree angle
                            change
1394                }
1395                else
1396                    dirVect = robot->GetGoal() - robot->GetPosition();
1397
1398                if(dirVect.z() > dirVect.y() && dirVect.z() > dirVect.x())//? or
                        greater than 6 or -6?
1399                    dirVect.zVal = 0;
1400
1401                if(! (dirVect.x() == 0 && dirVect.y() == 0) )
1402                {
1403                    robot->SetDirection(dirVect);
1404                    core::vector3df newCamTarget = camera[0]->getPosition() +
                            core::vector3df(dirVect.xVal, dirVect.zVal, dirVect.yVal)
                            ;
1405                    camera[0]->setTarget(newCamTarget);
1406                }
1407                else
1408                    int cat = 7;
1409            }
1410        }
1411        else
1412        {
1413            core::vector3df step = core::vector3df(shift.xVal/SCALE_FACTOR, shift
                    .zVal/SCALE_FACTOR, shift.yVal/SCALE_FACTOR);
1414            {
1415                float moveDist = MOVE_INCREMENTS_MM/SCALE_FACTOR;
1416                if(CalculateDirectDistance(robot->GetPosition(), robot->GetGoal()
                        , true)/SCALE_FACTOR < moveDist)//issue due to z coordinate.
1417                    moveDist = CalculateDirectDistance(robot->GetPosition(),
                            robot->GetGoal(), true)/SCALE_FACTOR;
1418                step = core::vector3df(shift.xVal, 0, shift.yVal).normalize();
1419                if(shift.xVal != 0)
1420                    step.Y = shift.zVal * step.X/shift.xVal;
1421                else if(shift.yVal != 0)
1422                    step.Y = shift.zVal * step.Z/shift.yVal;
1423                step = step * moveDist;
1424            }
1425
1426            robot->RecordMoreData("Desired_Steps.txt", step.X, step.Z, step.Y);//
                    * by scale?
1427
```

```
1428            if((int)(step.X * SCALE_FACTOR) == 0 && (int)(step.Z * SCALE_FACTOR)
                    == 0 && robot->NaviDone()) // (int)step.Y == 0 <-- don't know
                    height goal will be at when input it, so can't get equal to zero
                    unless alter/adapt after started running
1429            {
1430                goalReached = true;
1431                EndRun(goalReached, NULL);
1432            }
1433            else
1434            {
1435                robot->consecutiveTimesStationary = 0;
1436                Coord rPos = robot->GetPosition();
1437                core::vector3df newPos = (core::vector3df(rPos.x(), rPos.z(),
                        rPos.y()) + step*SCALE_FACTOR) / SCALE_FACTOR;
1438                int incrementalOffset = 100;
1439                int startLine[3] = {newPos.X, newPos.Y + incrementalOffset,
                        newPos.Z};
1440                int endLine[3] = {startLine[0], startLine[1] - 1, startLine[2]};
1441                float dist = CheckForCollision(startLine, endLine);
1442
1443                int watchDog = 0;
1444                while(dist == -1 && watchDog++ < 100)//seems to usually be due to
                         collision point being so close to current height value
1445                {
1446                    incrementalOffset += 100;
1447                    startLine[1] = newPos.Y + incrementalOffset;
1448                    endLine[1] = startLine[1] - 1;
1449                    dist = CheckForCollision(startLine, endLine);
1450                }
1451
1452                if(dist == -1)
1453                    incrementalOffset = 0;
1454
1455                float newHeight = (newPos.Y + incrementalOffset) - dist;
1456                newPos.Y = newHeight + ROBOTNODE_VERTBOOST;
1457                step.Y = newHeight - robot->GetPosition().zVal/SCALE_FACTOR;
1458                shift.zVal = step.Y * SCALE_FACTOR;
1459
1460                if(step.Y >= TANOFANGLE * MOVE_INCREMENTS_MM / SCALE_FACTOR ||
                        step.Y <= -TANOFANGLE * MOVE_INCREMENTS_MM / SCALE_FACTOR)
1461                {
1462                    //moving up or down steep slope.
1463                    Face_handle encompTri = Face_handle();
1464                    robot->terrain->GetTri(&encompTri, &robot->terrain->
                            GetCurrentNode()->GetPoint());
1465                    int angle = AngleBetweenTwoPlanes<Face_handle, Point>(
                            encompTri, NULL);
```

```
1466                int angle0 = AngleBetweenTwoPlanes<Face_handle, Point>(
                        encompTri->neighbor(0), NULL);
1467                int angle1 = AngleBetweenTwoPlanes<Face_handle, Point>(
                        encompTri->neighbor(1), NULL);
1468                int angle2 = AngleBetweenTwoPlanes<Face_handle, Point>(
                        encompTri->neighbor(2), NULL);
1469
1470                float dist[2] = {-1,-1};
1471                float dist0[2] = {-1,-1};
1472                float dist1[2] = {-1,-1};
1473                float dist2[2] = {-1,-1};
1474                CalculateDistsToCentre(*encompTri->GetPoint(0), *encompTri->
                        GetPoint(1), *encompTri->GetPoint(2), dist);
1475                CalculateDistsToCentre(*encompTri->neighbor(0)->GetPoint(0),
                        *encompTri->neighbor(0)->GetPoint(1), *encompTri->
                        neighbor(0)->GetPoint(2), dist0);
1476                CalculateDistsToCentre(*encompTri->neighbor(1)->GetPoint(0),
                        *encompTri->neighbor(1)->GetPoint(1), *encompTri->
                        neighbor(1)->GetPoint(2), dist1);
1477                CalculateDistsToCentre(*encompTri->neighbor(2)->GetPoint(0),
                        *encompTri->neighbor(2)->GetPoint(1), *encompTri->
                        neighbor(2)->GetPoint(2), dist2);
1478                if((angle > SLOPEANGLE && angle < 180 - SLOPEANGLE && dist[0]
                         < HAZARD_TRI_DIST_THRESHOLD) || (angle0 > SLOPEANGLE &&
                        angle0 < 180 - SLOPEANGLE && dist0[0] <
                        HAZARD_TRI_DIST_THRESHOLD) || (angle1 > SLOPEANGLE &&
                        angle1 < 180 - SLOPEANGLE && dist1[0] <
                        HAZARD_TRI_DIST_THRESHOLD) || (angle2 > SLOPEANGLE &&
                        angle2 < 180 - SLOPEANGLE && dist2[0] <
                        HAZARD_TRI_DIST_THRESHOLD))
1479                {
1480                    printf("Attempted to cross slope which is too steep. Rise
                            of %d, angles of %d\n", (int) step.Y, angle);
1481                    printf("Angles of neighbours are %d %d %d\n", angle0,
                            angle1, angle2);
1482                    EndRun(goalReached, "Simulation ended as robot tried to
                            traverse hazard. ");
1483                }
1484                else
1485                {
1486                    if(robot->terrain->GetTri(&encompTri, &Point(newPos.X,
                            newPos.Z, newPos.Y)))
1487                    {
1488                        angle = AngleBetweenTwoPlanes<Face_handle, Point>(
                                encompTri, NULL);
1489                        angle0 = AngleBetweenTwoPlanes<Face_handle, Point>(
                                encompTri->neighbor(0), NULL);
```

```
1490                        angle1 = AngleBetweenTwoPlanes<Face_handle, Point>(
                                encompTri->neighbor(1), NULL);
1491                        angle2 = AngleBetweenTwoPlanes<Face_handle, Point>(
                                encompTri->neighbor(2), NULL);
1492                        CalculateDistsToCentre(*encompTri->GetPoint(0), *
                                encompTri->GetPoint(1), *encompTri->GetPoint(2),
                                dist);
1493                        CalculateDistsToCentre(*encompTri->neighbor(0)->
                                GetPoint(0), *encompTri->neighbor(0)->GetPoint(1)
                                , *encompTri->neighbor(0)->GetPoint(2), dist0);
1494                        CalculateDistsToCentre(*encompTri->neighbor(1)->
                                GetPoint(0), *encompTri->neighbor(1)->GetPoint(1)
                                , *encompTri->neighbor(1)->GetPoint(2), dist1);
1495                        CalculateDistsToCentre(*encompTri->neighbor(2)->
                                GetPoint(0), *encompTri->neighbor(2)->GetPoint(1)
                                , *encompTri->neighbor(2)->GetPoint(2), dist2);
1496                        if((angle > SLOPEANGLE && angle < 180 - SLOPEANGLE &&
                                 dist[0] < HAZARD_TRI_DIST_THRESHOLD) || (angle0
                                > SLOPEANGLE && angle0 < 180 - SLOPEANGLE &&
                                dist0[0] < HAZARD_TRI_DIST_THRESHOLD) || (angle1
                                > SLOPEANGLE && angle1 < 180 - SLOPEANGLE &&
                                dist1[0] < HAZARD_TRI_DIST_THRESHOLD) || (angle2
                                > SLOPEANGLE && angle2 < 180 - SLOPEANGLE &&
                                dist2[0] < HAZARD_TRI_DIST_THRESHOLD))
1497                        {
1498                            printf("Attempted␣to␣cross␣slope␣which␣is␣too␣
                                    steep.␣Rise␣of␣%d,␣angles␣of␣%d\n", (int)
                                    step.Y, angle);
1499                            printf("Angles␣of␣neighbours␣are␣%d␣%d␣%d\n",
                                    angle0, angle1, angle2);
1500                            EndRun(goalReached, "Simulation␣ended␣as␣robot␣
                                    tried␣to␣traverse␣hazard.␣");
1501                        }
1502                    }
1503                }
1504            }
1505
1506            robot->RecordMoreData("Actual_Steps.txt", step.X, step.Z, step.Y)
                    ;
1507            printf("Moved␣%d␣%d␣%d\n", (int) step.X, (int) step.Z, (int) step
                    .Y);
1508
1509            robot->AdjustPosition((int)(step.X * SCALE_FACTOR), (int)(step.Z
                    * SCALE_FACTOR), (int)(step.Y * SCALE_FACTOR));//why use step
                     over shift?
1510            robotNode->setPosition(newPos);
```

```
1511                core::vector3df newCamPos = newPos + core::vector3df(0,
                        CAMERA_VERTBOOST,0);
1512                core::vector3df newCamTarget = newPos + step + core::vector3df(0,
                        CAMERA_VERTBOOST,0);
1513                newCamTarget.Y = newCamPos.Y; //To have camera looking
                        horizontally, ie rather than skyward when climbing a rise
1514                camera[0]->setPosition(newCamPos);
1515                camera[0]->setTarget(newCamTarget);
1516                robot->SetDirection(shift); //adjust vert of new view
1517            }
1518        }
1519
1520        return goalReached;
1521    }
1522
1523    void SimulationEnvironment::CloseEnvironment()
1524    {
1525        Coord rPos = robot->GetPosition();
1526        robot->RecordEndOfData("Desired_Steps.txt", rPos.xVal, rPos.yVal, rPos.
                zVal);
1527        robot->RecordEndOfData("Actual_Steps.txt", rPos.xVal, rPos.yVal, rPos.
                zVal);
1528        robot->RecordEndOfData("Way_Points.txt", rPos.xVal, rPos.yVal, rPos.zVal)
                ;
1529        delete robot;
1530        robot = NULL;
1531
1532        free(anim);
1533        device->drop();
1534    }
1535
1536    scene::ITerrainSceneNode* SimulationEnvironment::GetTerrain()
1537    {
1538        return terrain;
1539    }
1540
1541    IrrlichtDevice* SimulationEnvironment::GetDevice()
1542    {
1543        return device;
1544    }
1545
1546    scene::ICameraSceneNode* SimulationEnvironment::GetCamera(int num)
1547    {
1548        if(num >= 0 && num < 3)
1549            return camera[num];
1550        else
1551            return NULL;
```

```
1552    }
1553
1554    void SimulationEnvironment::ToggleMinMax()
1555    {
1556    }
1557
1558    void SimulationEnvironment::ToggleSimRun()
1559    {
1560        runSim = !runSim;
1561    }
1562
1563    void SimulationEnvironment::ToggleRobotRun()
1564    {
1565        runRobot = !runRobot;
1566    }
1567
1568    void SimulationEnvironment::ToggleActiveWindowMode()
1569    {
1570        allowInactiveScreen = !allowInactiveScreen;
1571        gui::IGUIElement* rootElement = env->getRootGUIElement();
1572
1573        rootElement->removeChild(rootElement->getElementFromId(GUI_ACT_WIN_MODE,
                false));
1574        if(allowInactiveScreen)
1575            env->addStaticText(L"Sim runs when window is: Active or Inactive",
                    core::rect<s32>(GAP + COLUMN_WIDTH + GAP + GAP, VIEW_HEIGHT + GAP
                    , GAP + COLUMN_WIDTH + GAP + GAP + COLUMN_WIDTH, VIEW_HEIGHT +
                    GAP + LINE_HEIGHT + LINE_HEIGHT), true, true, 0, GUI_ACT_WIN_MODE
                    , true);
1576        else
1577            env->addStaticText(L"Sim runs when window is: Active Only", core::
                    rect<s32>(GAP + COLUMN_WIDTH + GAP + GAP, VIEW_HEIGHT + GAP, GAP
                    + COLUMN_WIDTH + GAP + GAP + COLUMN_WIDTH, VIEW_HEIGHT + GAP +
                    LINE_HEIGHT + LINE_HEIGHT), true, true, 0, GUI_ACT_WIN_MODE, true
                    );
1578    }
1579
1580    void SimulationEnvironment::ToggleSkies()
1581    {
1582        showBox = !showBox;
1583        skybox->setVisible(showBox);
1584        skydome->setVisible(!showBox);
1585    }
1586
1587    void SimulationEnvironment::DisplayGraphLinks()
1588    {
1589        robot->terrain->GetAllLinks();
```

```
1590        display("AllLinks.txt");
1591    }
1592
1593    void SimulationEnvironment::DisplayPath()
1594    {
1595        robot->DisplayPlannedPath();
1596    }
1597
1598
1599    void const SimulationEnvironment::SwitchView()
1600    {
1601        switch(++view%3)
1602        {
1603            case 0: smgr->setActiveCamera(camera[0]);break;
1604            case 1: smgr->setActiveCamera(camera[1]);break;
1605            case 2: smgr->setActiveCamera(camera[2]);break;
1606            default: ; //fault
1607        }
1608    }
1609
1610    void const SimulationEnvironment::SwitchView(int camNum)
1611    {
1612        if(camNum >= 0 && camNum < 3)
1613        {
1614            smgr->setActiveCamera(camera[camNum]);
1615            view = camNum;
1616        }
1617        else
1618            camNum = 0;
1619    }
1620
1621    bool SimulationEnvironment::Exited()
1622    {
1623        bool result = false;
1624        if(camera == NULL)
1625            result = true;
1626
1627        return result;
1628    }
1629
1630    void SimulationEnvironment::ExitSim()
1631    {
1632        free(camera);
1633        camera = NULL;
1634        device->closeDevice();
1635    }
1636
```

```
1637    void SimulationEnvironment::EndRun()
1638    {
1639        device->closeDevice();
1640    }
1641
1642    void SimulationEnvironment::EndRun(bool goalReached, char* message)
1643    {
1644        FILE* output = fopen("SimResults.txt", "at");
1645        if(message != NULL)
1646            fprintf(output, message);
1647
1648        if(robot->consecutiveTimesStationary == 10)
1649        {
1650            robot->terrain->CullHeldNodes();
1651            if(!robot->terrain->GoalReachable())
1652            {
1653                fprintf(output, "Goal node was not reachable due to being within
                        a hazardous region. ");
1654            }
1655            if(robot->terrain->GetNumHeldNodes() < 2)//ie only goal node left
1656            {
1657                fprintf(output, "HeldNodes only contained goal at close. ");
1658            }
1659        }
1660
1661        if(goalReached)
1662            fprintf(output, "Simulation successfully reach the goal\n");
1663        else
1664            fprintf(output, "Simulation ended without reaching the goal\n");
1665
1666        fclose(output);
1667        device->closeDevice();
1668    }
1669
1670    void SimulationEnvironment::ScanViewPoints(Coord** terrainPoints, int *
            numPoints, Coord position, Coord direction)
1671    {
1672        core::line3d<f32> lineForScan, line;
1673        core::vector3df intersectionForScan, targetDirection, xNorm, yNorm;
1674        core::triangle3df triForScan;
1675
1676        lineForScan.start = core::vector3df(position.xVal/SCALE_FACTOR, position.
                zVal/SCALE_FACTOR + CAMERA_HEIGHT, position.yVal/SCALE_FACTOR);
1677        targetDirection = (core::vector3df(direction.xVal/SCALE_FACTOR, direction
                .zVal/SCALE_FACTOR, direction.yVal/SCALE_FACTOR)).normalize();
1678        Coord normVectX = CrossProduct(direction, direction + Coord(0,0,10));
1679        Coord normVectY = CrossProduct(direction, direction + Coord(10,0,0));
```

```
1680        xNorm = (core::vector3df(normVectX.xVal/SCALE_FACTOR, normVectX.zVal/
                SCALE_FACTOR, normVectX.yVal/SCALE_FACTOR)).normalize();
1681        yNorm = (core::vector3df(normVectY.xVal/SCALE_FACTOR, normVectY.zVal/
                SCALE_FACTOR, normVectY.yVal/SCALE_FACTOR)).normalize();
1682
1683        int index = 0;
1684        const int xArraySize = 21, yArraySize = 21;
1685        float xArray[xArraySize] = {-0.5, -0.45, -0.4, -0.35, -0.3, -0.25, -0.2,
                -0.15, -0.1, -0.05, 0,
1686                0.5, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5};
1687        float yArray[yArraySize] = {-2, -1, -0.666, -0.5, - 0.4, -0.333, -.285,
                -0.25, -0.222, -0.2,
1688                 -0.167, -0.143, -0.125, -0.111 -0.1, 0, 0.1, 0.2, 0.25,
                        0.333, 0.5};
1689
1690        for(int i = 0; i < yArraySize; i++)
1691        {
1692            for(int j = 0; j < xArraySize; j++)
1693            {
1694                float x = xArray[j];
1695                float y = yArray[i];
1696
1697                lineForScan.end = lineForScan.start + (targetDirection + xNorm*x
                        + yNorm*y) * (VISIBLE_DIST_MM / SCALE_FACTOR);
1698
1699                if(smgr->getSceneCollisionManager()->getCollisionPoint(
                        lineForScan, metaSelector, intersectionForScan, triForScan))
1700                {
1701                    if(index != 0 && index%ARRAY_CHUNK == 0)
1702                    {
1703                        void* tmp = realloc(*terrainPoints, (index + ARRAY_CHUNK)
                                * sizeof(Coord));
1704                        if(tmp != NULL)
1705                            *terrainPoints = (Coord*) tmp;
1706                    }
1707
1708                    (*terrainPoints)[index].xVal = (int) intersectionForScan.X *
                            SCALE_FACTOR;
1709                    (*terrainPoints)[index].yVal = (int) intersectionForScan.Z *
                            SCALE_FACTOR; //The Y & Z coords in irrlicht are reverse
                            to robot
1710                    (*terrainPoints)[index++].zVal = (int) intersectionForScan.Y
                            * SCALE_FACTOR;
1711                }
1712                else
1713                    if(SHOW_SIM_WARNINGS) printf("Warning: ***** No point of
                            collision ***** \n");
```

```
1714            }
1715        }
1716        *numPoints = index;
1717
1718        if(index == 0)
1719        {
1720            if(SHOW_SIM_ERRORS) printf("Error: ********* No points of collision
                ********* \n");
1721        }
1722
1723        return;
1724    }
1725
1726    /** Location errors are added through a difference between realPos and
        robotPos
1727    * Orientation errors could be added but require work change/calc of
        targetDirection and then changing when added to create new point - ie
        work out distance time x and y components at diff angle
1728    */
1729    void SimulationEnvironment::ScanViewPoints_v2(Coord** terrainPoints, int *
        numPoints)
1730    {
1731        Coord robotPos = robot->GetPosition();
1732        Coord direction = robot->GetDirection();
1733        Coord normVectX = CrossProduct(direction, direction + Coord(0,0,10));
1734        Coord normVectY = CrossProduct(direction, direction + Coord(10,0,0));
1735
1736        core::line3d<f32> lineForScan, line;
1737        core::vector3df intersectionForScan, xNorm, yNorm;
1738        core::triangle3df triForScan;
1739        core::vector3df realPos = robotNode->getPosition();
1740        core::vector3df targetDirection;// = robotNode->getPosition() + robotNode
            ->getRotation();
1741
1742        lineForScan.start = realPos + core::vector3df(0, CAMERA_HEIGHT, 0);
1743        targetDirection = (core::vector3df(direction.xVal/SCALE_FACTOR, direction
            .zVal/SCALE_FACTOR, direction.yVal/SCALE_FACTOR)).normalize();
1744
1745        xNorm = (core::vector3df(normVectX.xVal/SCALE_FACTOR, normVectX.zVal/
            SCALE_FACTOR, normVectX.yVal/SCALE_FACTOR)).normalize();
1746        yNorm = (core::vector3df(normVectY.xVal/SCALE_FACTOR, normVectY.zVal/
            SCALE_FACTOR, normVectY.yVal/SCALE_FACTOR)).normalize();
1747
1748        int index = 0;
1749        const int xArraySize = 21, yArraySize = 21;
1750        float xArray[xArraySize] = {-0.5, -0.45, -0.4, -0.35, -0.3, -0.25, -0.2,
            -0.15, -0.1, -0.05, 0,
```

```
1751            0.5, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5};
1752        float yArray[yArraySize] = {-2, -1, -0.666, -0.5, - 0.4, -0.333, -.285,
                -0.25, -0.222, -0.2,
1753            -0.167, -0.143, -0.125, -0.111 -0.1, 0, 0.1, 0.2, 0.25, 0.333, 0.5};
1754
1755        for(int i = 0; i < yArraySize; i++)
1756        {
1757            for(int j = 0; j < xArraySize; j++)
1758            {
1759                float x = xArray[j];
1760                float y = yArray[i];
1761
1762                lineForScan.end = lineForScan.start + (targetDirection + xNorm*x
                        + yNorm*y) * (VISIBLE_DIST_MM / SCALE_FACTOR);
1763
1764                if(smgr->getSceneCollisionManager()->getCollisionPoint(
                        lineForScan, metaSelector, intersectionForScan, triForScan))
1765                {
1766                    if(index != 0 && index%ARRAY_CHUNK == 0)
1767                    {
1768                        void* tmp = realloc(*terrainPoints, (index + ARRAY_CHUNK)
                                * sizeof(Coord));
1769                        if(tmp != NULL)
1770                            *terrainPoints = (Coord*) tmp;
1771                    }
1772
1773                    float noise = 1;
1774                    if(INCLUDE_NOISE)
1775                        noise += (rand()%101) / 100 * MAX_NOISE;
1776
1777                    //newPoint = believe positions + distance measured between
                            real position and surface/intersection
1778                    (*terrainPoints)[index].xVal = robotPos.xVal + (int) ((
                            realPos.X - intersectionForScan.X) * SCALE_FACTOR * noise
                            );
1779                    (*terrainPoints)[index].yVal = robotPos.yVal + (int) ((
                            realPos.Z - intersectionForScan.Z) * SCALE_FACTOR * noise
                            );//The Y & Z coords in irrlicht are reverse to robot
1780                    (*terrainPoints)[index++].zVal = robotPos.zVal + (int) ((
                            realPos.Y - intersectionForScan.Y) * SCALE_FACTOR * noise
                            );
1781                }
1782                else
1783                    if(SHOW_SIM_WARNINGS) printf("Warning: ***** No point of
                            collision ***** \n");
1784            }
1785        }
```

```
1786            *numPoints = index;
1787
1788            if(index == 0)
1789            {
1790                if(SHOW_SIM_ERRORS) printf("Error: ********* No points of collision
                        ********* \n");
1791            }
1792
1793            return;
1794        }
1795
1796        void SimulationEnvironment::PanView()
1797        {
1798            Coord dirVect = NormaliseVector<Coord>(robot->GetDirection(),
                    MOVE_INCREMENTS_MM/SCALE_FACTOR, false);
1799            dirVect.zVal = 0;
1800            Coord normVect = ScaledCrossProduct(dirVect, dirVect + Coord(0,0,
                    MOVE_INCREMENTS_MM/SCALE_FACTOR), MOVE_INCREMENTS_MM/SCALE_FACTOR);
1801
1802            Coord* scannedData = (Coord*) calloc(ARRAY_CHUNK, sizeof(Coord));
1803            int numScanPoints = 0;
1804
1805            ScanViewPoints(&scannedData, &numScanPoints, robot->GetPosition(),
                    dirVect);
1806            robot->GatherData(scannedData, numScanPoints);
1807
1808            numScanPoints = 0;
1809            ScanViewPoints(&scannedData, &numScanPoints, robot->GetPosition(),
                    dirVect + normVect);
1810            robot->GatherData(scannedData, numScanPoints);
1811
1812
1813            numScanPoints = 0;
1814            ScanViewPoints(&scannedData, &numScanPoints, robot->GetPosition(),
                    dirVect - normVect);
1815            robot->GatherData(scannedData, numScanPoints);
1816
1817            free(scannedData);
1818        }
1819
1820        float SimulationEnvironment::CheckForCollision(int start[3], int end[3])
1821        {
1822            core::line3d<f32> line;
1823            core::vector3df intersect;
1824            core::triangle3df tri;
1825            float dist = -1;
1826
```

```
1827            line.start = core::vector3df(start[0],start[1],start[2]);
1828            line.end = line.start + (core::vector3df(end[0],end[1],end[2]) - line.
                    start).normalize() * 5000.0f;
1829
1830            if(smgr->getSceneCollisionManager()->getCollisionPoint(line, metaSelector
                    , intersect, tri))
1831            {
1832                if(!(intersect.X - start[0] > end[0] - start[0] && intersect.Y -
                        start[1] > end[1] - start[1] && intersect.Z - start[2] > end[2] -
                         start[2]))
1833                {
1834                    dist = sqrt((intersect.X - start[0])*(intersect.X - start[0]) + (
                        intersect.Y - start[1])*(intersect.Y - start[1]) + (intersect
                        .Z - start[2])*(intersect.Z - start[2]));
1835                }
1836            }
1837
1838            return dist;
1839        }
1840
1841        float SimulationEnvironment::CheckForCollision(core::vector3df start)
1842        {
1843            core::line3d<f32> line;
1844            core::vector3df intersect;
1845            core::triangle3df tri;
1846            float dist = -1;
1847
1848            line.start = start;
1849            line.end = line.start - core::vector3df(0,5000.0f,0);
1850
1851            if(smgr->getSceneCollisionManager()->getCollisionPoint(line, metaSelector
                    , intersect, tri))
1852            {
1853                if(!(intersect.X - start.X > 0 && intersect.Y - start.Y > 1 &&
                        intersect.Z - start.Z > 0))
1854                {
1855                    dist = sqrt((intersect.X - start.X)*(intersect.X - start.X) + (
                        intersect.Y - start.Y)*(intersect.Y - start.Y) + (intersect.Z
                         - start.Z)*(intersect.Z - start.Z));
1856                }
1857            }
1858
1859            return dist;
1860        }
```

A.8 World

A.8.1 World.h

```
1  #ifndef WORLD_H_
2  #define WORLD_H_
3
4  #include <typeinfo>
5  #include "Hier_Triangulation.h"
6  #include "Nodes.h"
7
8
9  typedef Hierarchy2DMesh::Vertex_handle   Vertex_handle;
10 typedef Hierarchy2DMesh::Face_handle     Face_handle;
11 typedef Hierarchy2DMesh::Vector       Vector;
12 typedef Hierarchy2DMesh::Point        Point;
13
14 enum TRI_MODE{
15     HAZ_ITER = 0,
16     HAZ_RECURS,
17     HIER_2,
18     HIER_3
19 };
20
21 const bool SHOW_SDL = true;
22 const bool SHOW_INFO = true;
23 const bool SHOW_ERRORS = true;
24 const bool SHOW_WARNINGS = false;
25 #define SCALE_FACTOR 10
26
27 const int MEASUREMENT_PRECISION = 50; // difference in height(mm) over which z-
      measurement is guaranteed to be added
28 const int COPLANAR_PRECISION = 5; // num mm a z-measurement must be within to be
      considered coplanar
29 const int DIST_THRESHOLD = 200; //min distance needed between new point and one
      of old points of its encompassing tri
30 //ie even if is really large tri, new point is only added if it is more than
      dist_threshold from all the vertices
31 const int COPLANAR_DIST_THRESHOLD = 400; //min distance needed between new point
      and one of old points of its encompassing tri
32 const int UNKNOWN_DIST_THRESHOLD = 400; //min distance needed between new point
      and one of two points from the inf tri encompassing it
33 const int HAZARD_TRI_DIST_THRESHOLD = 600; //Max dist between centre and vertex a
       hazard tri can be and still reliably trust
34 const int MAPNODE_PROXIMITY_THRESHOLD = 100; // -- affects detecting hazard is in
       movement path?
```

```
35 const int MAX_LINK_DISTANCE = 10000;//To limit links which cross great distance,
       when could go via intermediaries
36 //visible distance is 10,000 and hence radius of current mapNodes, with key nodes
       being vis_dist * key_ratio ie 14000
37
38 const float KEY_RATIO = 1.414;//1.41?
39 const int MIN_INCIDENT_ANGLE = 10;
40 const Point ZERO_POINT = Point(0,0,0);
41
42 class Hazard{
43 public:
44     Hazard();
45     Hazard(Point* hPoints, HAZARDS hType);
46     void AddPoint(Point newPt);
47     void AdjoinHazard(Hazard* otherHazard);
48     bool CompareHazard(Hazard* otherHazard);
49     int GetNumPoints() { return numPoints; }
50     Point* GetPoints() { return points; }
51     HAZARDS GetHazardType() { return hazardType; }
52 private:
53     Point* points;
54     int numPoints;
55     HAZARDS  hazardType;//necessary or useful?
56 };
57
58 template <class NODE>
59 class World{
60 public:
61     virtual ~World();
62     World(Coord gpsCoords, Coord loc, Coord dest);
63     void SetHazardIdentParam(int sAngle, int wAngle, int rLength, int rWidth, int
           aSep, int wRadius, int wWidth, int cBuffer, float mDist, float vDist);
64     void LoadKnown(bool nodesKnown);
65     void AllocateNodeArrays();
66     void FreeNodeArrays();
67
68     bool GoalReachable();
69     void AdjustLocation(int x, int y, int z);
70     void AddPointsToWorld(Coord *newData, int numCoords);
71     void ProcessPoints(bool known);
72     void GenHazardNodes(HazardNode** hNodes, int* numNodes, Point dirVect);
73     void GenMapNodes(HazardNode* hNodes, int numNodes, Point dirVect);
74     void GenPreMadeMapNodes(Point dirVect);
75     void GenGridNodes(Point dirVect);
76
77     void CullHeldNodes();
78     void GetAllLinks();
```

```
79      void GetAllKeyNodeLinks();
80      int GetNumHeldNodes() { return numHeldNodes; }
81
82      bool CheckBetweenPoints(Point start, Point end, bool checkParallel, Point
            normVect);
83
84      NODE* GetCurrentNode();
85      NODE* GetGoalNode();
86
87      void GarbageCollect();
88      float CalculateUndulatingDistance(Point start, Point end);//obsolete?
89      bool GetTri(Triangle** tri, Coord *point);//Obsolete, remove along with
            CheckEdgePath from Edgehugger
90      bool GetTri(Face_handle* tri, Point *point);
91      void GetAllTriangles(Triangle*** allTriangles, int *numTri);
92      void GetAllTriangles(Face_handle** allTriangles, int *numTri);
93
94      /** For incremental/life long path planning **/
95      NODE** closedSet;
96      NODE** openSet;
97      int openInd;
98      int closedInd;
99      /** ***** **/
100 private:
101     void AddPoint(Coord* newPoint);
102     void AddHazard(Hazard hazard);
103     void DiscardPoints();
104     bool RecursExpandHazardObject(Hazard *newHazard, Face_handle hazTri);
105
106     bool CheckBetweenHazardNodes(Point pt1, Point pt2, bool* isSlope);
107     void CullKeyNodes(NODE** mNodes, int* numMapNodes, Point dirVect);
108     void CullNodeSets();
109     void CreateGrid();
110     void ExpandNode(NODE* edgeNode, int eInd);
111     void CullGridNodes(Point dirVect);
112     void FreeMarkedNodes();
113     void LinkMapNodes(NODE** mNodes, int* numMapNodes);
114     void LinkKeyNodes(NODE** mNodes, int* numMapNodes, Point dirVect);
115     void SortNodes(NODE** mNodes, int* numMapNodes);
116     void MergeWithKeyNodes(NODE** mNodes, int* numMapNodes, bool preSorted);
117     void LinkOtherNodes();
118
119     template <class TRI, class PT> void IdentifyHazards(int numTri, TRI* surface)
            ;
120     template <class TRI, class PT> void IdentifyKnownHazards(int numTri, TRI*
            surface);
```

```
121     template <class TRI, class PT> bool CheckWall(TRI* current, TRI* testTri,
            HAZARDS* hazType);
122
123     Hierarchy2DMesh hMesh2D;
124
125     Coord gpsCoords; /**[3]GPS xyz co-ordinates which World is relative to*/
126     Coord destination;
127     Coord location;
128
129     bool arraysInitialised;
130     bool graphInitialised;
131
132     Point startPt;
133     NODE* currentLocNode;
134     NODE* goalNode;
135
136     NODE** heldNodes; /** list of Nodes */
137     int numHeldNodes; /** num nodes in list */
138     NODE** keyNodes; /** ptrs to close Nodes */
139     int numKeyNodes; /** num close nodes in list */
140     NODE** edgeNodes; /** ptrs to edge Nodes */
141     int numEdgeNodes; /** num edge nodes in list */
142
143     int numPreMadePoints;
144     bool *preMadeAdded;
145     Point *preMadePoints;
146     int numPreMadeHazards;
147     Point* preKnownHazards;
148
149     Hazard** savedHazards;
150     int numSavedHazards;
151     Hazard** borderHazards;
152     int numBorderHazards;
153
154     int SLOPEANGLE;
155     int WALLANGLE;
156     int ROBOTLENGTH;
157     int ROBOTWIDTH;
158     int AXLESEP; //distance between axles of robot
159     int WHEELWIDTH;
160     int WHEELRADIUS;
161     int CLEARANCEBUFFER;
162     int MOVE_INCR;
163     float VISIBLE_DIST_MM;
164 };
165
166     template<class NODE>
```

```
167     World<NODE>::World(Coord refCoords, Coord loc, Coord dest)
168     {
169         location = loc;
170         destination = dest;
171         gpsCoords = refCoords;
172         startPt = Point(location.x(), location.y(), location.z());
173 
174         closedSet = (NODE**) calloc(ARRAY_CHUNK, sizeof(NODE*));
175         openSet = (NODE**) calloc(ARRAY_CHUNK, sizeof(NODE*));
176         openInd = 0;
177         closedInd = 0;
178 
179         heldNodes = (NODE**) calloc(ARRAY_CHUNK, sizeof(NODE*));
180         heldNodes[0] = new NODE(destination.x(), destination.y(), destination.z()
                );
181         heldNodes[1] = new NODE(location.x(), location.y(), location.z());
182         numHeldNodes = 2;
183         goalNode = heldNodes[0]; //goalNode is always zero
184         currentLocNode = new NODE(heldNodes[1]->GetPoint());
185         keyNodes = (NODE**) calloc(ARRAY_CHUNK, sizeof(NODE*));
186         keyNodes[0] = goalNode;
187         numKeyNodes = 1;
188         edgeNodes = (NODE**) calloc(ARRAY_CHUNK, sizeof(NODE*));
189         numEdgeNodes = 0;
190 
191         savedHazards = (Hazard**) calloc(ARRAY_CHUNK, sizeof(Hazard*));
192         numSavedHazards = 0;
193         borderHazards = (Hazard**) calloc(LIST_CHUNK, sizeof(Hazard*));
194         numBorderHazards = 0;
195 
196         arraysInitialised = true;
197         graphInitialised = false;
198 
199         numPreMadePoints = 0;
200         preMadeAdded = NULL;
201         preMadePoints = NULL;
202         numPreMadeHazards = 0;
203         preKnownHazards = NULL;
204     }
205 
206     template<class NODE>
207     World<NODE>::~World()
208     {
209         for(int i = 0; i < numHeldNodes; i++)
210         {
211             if(heldNodes[i] == NULL)
212                 continue;
```

```
213
214             delete heldNodes[i];
215             heldNodes[i] = NULL;
216         }
217
218         free(openSet);
219         openSet = NULL;
220         free(closedSet);
221         closedSet = NULL;
222         free(keyNodes);
223         keyNodes = NULL;
224         free(heldNodes);
225         heldNodes = NULL;
226
227         delete goalNode;
228         goalNode = NULL;
229         delete currentLocNode;
230         currentLocNode = NULL;
231
232         free(savedHazards);
233         savedHazards = NULL;
234         free(borderHazards);
235         borderHazards = NULL;
236
237         free(preMadeAdded);
238         free(preMadePoints);
239         free(preKnownHazards);
240         preMadeAdded = NULL;
241         preMadePoints = NULL;
242         preKnownHazards = NULL;
243     }
244
245     template<class NODE>
246     void World<NODE>::AdjustLocation(int x, int y, int z)
247     {
248         location.xVal += x;
249         location.yVal += y;
250         location.zVal += z;
251         delete(currentLocNode);
252         currentLocNode = new NODE(ConvertPointType<Coord, Point>(location));
253     }
254
255     template<class NODE>
256     void World<NODE>::AllocateNodeArrays()
257     {
258         if(!arraysInitialised)
259         {
```

```
260             heldNodes = (NODE**) calloc(ARRAY_CHUNK, sizeof(NODE*));
261             heldNodes[0] = new NODE(destination.x(), destination.y(), destination
                    .z());
262             heldNodes[1] = new NODE(location.x(), location.y(), location.z());
263             numHeldNodes = 2;
264             goalNode = heldNodes[0];
265             keyNodes = (NODE**) calloc(ARRAY_CHUNK, sizeof(NODE*));
266             keyNodes[0] = goalNode;
267             numKeyNodes = 1;
268             edgeNodes = (NODE**) calloc(ARRAY_CHUNK, sizeof(NODE*));
269             numEdgeNodes = 0;
270             arraysInitialised = true;
271         }
272     }
273
274     template<class NODE>
275     void World<NODE>::FreeNodeArrays()
276     {
277         if(arraysInitialised)
278         {
279             for(int i = 0; i < numHeldNodes; i++)
280             {
281                 if(heldNodes[i] == NULL)
282                     continue;
283
284                 delete heldNodes[i];
285                 heldNodes[i] = NULL;
286             }
287
288             free(heldNodes);
289             free(keyNodes);
290             heldNodes = NULL;
291             keyNodes = NULL;
292             currentLocNode = NULL;
293             goalNode = NULL;
294             arraysInitialised = false;
295         }
296     }
297
298     template<class NODE>
299     bool World<NODE>::GoalReachable()
300     {
301
302         bool result = true;
303         if(CalculateDirectDistance(currentLocNode->GetPoint(), goalNode->GetPoint
                ()), true) < VISIBLE_DIST_MM * KEY_RATIO)
304         {
```

```
305             Face_handle encompTri = Face_handle();
306             ///If neighbours are hazards does that mean is unreachable ?
307             /// If any neighbour is hazard, prob means goal is within clearance
                    of hazard ie bad/unreachable
308             if(GetTri(&encompTri, &goalNode->GetPoint()) && (encompTri->
                    hazardType || (encompTri->neighbor(0)->hazardType && encompTri->
                    neighbor(1)->hazardType && encompTri->neighbor(2)->hazardType)))
309             {
310                 //can't reach goal
311                 FILE* output = fopen("SimResults.txt", "at");
312                 float dist = CalculateDirectDistance(currentLocNode->GetPoint(),
                        goalNode->GetPoint(), true);
313                 fprintf(output, "Robot was %5.2f away from goal when concluded
                        was not reachable. ", dist);
314                 fclose(output);
315                 result = false;
316             }
317         }
318
319         return result;
320     }
321
322     /**
323     Add points to XYMatrix, for every new data point increment newPointCount or
            newFakePointCount
324     - If a number of points added are located close together, call identifyHazard
            on that region,
325         also interpolate
326     - If many points added, then call identifyHazards on entire MapSection, also
            call interpolation
327     */
328     template<class NODE>
329     void World<NODE>::AddPointsToWorld(Coord *newData, int numCoords)
330     {
331         if(SHOW_INFO) printf("Number of new data points %d \n", numCoords);
332
333         if(newData != NULL && numCoords > 0)
334         {
335             for(int i = 0; i < numCoords; i++)
336             {
337                 AddPoint(&newData[i]);
338             }
339
340             Face_handle* allTri = (Face_handle*) calloc(ARRAY_CHUNK, sizeof(
                    Face_handle));
341             int numTri = 0;
342             GetAllTriangles(&allTri, &numTri);
```

```
343             WriteTriToFile("mesh_triangles.txt", "wt", allTri, numTri, false);
344
345             FILE* output = fopen("/home/haz/NumTriInTessellation.txt", "at");
346             fprintf(output, "Number of triangles in this tessellation: %i\n",
                    numTri);
347             fclose(output);
348             free(allTri);
349             allTri = NULL;
350         }
351     }
352
353     template<class NODE>
354     float World<NODE>::CalculateUndulatingDistance(Point start, Point end)
355     {
356         float distance = 0;
357         Point dirVector = end - (start - Point(0,0,0));
358         bool triExists = false;
359         Point lineStart = start, lineEnd;
360         Face_handle encompTri = Face_handle();
361
362         while(CalculateDirectDistance(lineStart, end, true) > 15)
363         {
364             if(GetTri(&encompTri, &lineStart) && InsideTriangleTest(lineStart, *
                    encompTri->GetPoint(0), *encompTri->GetPoint(1), *encompTri->
                    GetPoint(2)) == INSIDE)
365             {
366                 if(encompTri->hazardType)
367                 {
368                     printf("Error - due to buffer, should never get to a tri
                            which is hazard even if is on edge\n");
369                     break;
370                 }
371
372                 if(GetIntersection(*encompTri->GetPoint(0), *encompTri->GetPoint
                        (1), lineStart, end, &lineEnd))
373                     distance += CalculateDirectDistance(lineStart, lineEnd, false
                            );
374                 else if(GetIntersection(*encompTri->GetPoint(0), *encompTri->
                        GetPoint(2), lineStart, end, &lineEnd))
375                     distance += CalculateDirectDistance(lineStart, lineEnd, false
                            );
376                 else if(GetIntersection(*encompTri->GetPoint(1), *encompTri->
                        GetPoint(2), lineStart, end, &lineEnd))
377                     distance += CalculateDirectDistance(lineStart, lineEnd, false
                            );
378                 else if(InsideTriangleTest(end, *encompTri->GetPoint(0), *
                        encompTri->GetPoint(1), *encompTri->GetPoint(2)))
```

```
379                 {
380                     lineEnd = end;
381                     distance += CalculateDirectDistance(lineStart, end, false);
382                 }
383             }
384             else
385             {
386                 lineEnd = lineStart + (NormaliseVector(dirVector, 50.0f, false) -
                        Point(0,0,0));// or should this be to centre of tri
387                 distance += CalculateDirectDistance(lineStart, lineEnd, false);
388             }
389 
390             lineStart = lineEnd + (NormaliseVector(dirVector, 10.0f, false) -
                    Point(0,0,0));
391             distance += 10;//to account for shift past lineEnd to put into next
                    tri
392         }
393 
394         return distance;
395     }
396 
397     template<class NODE>
398     bool World<NODE>::GetTri(Triangle** tri, Coord *point)
399     {
400         bool result = false;
401         (*tri)->point1 = new Coord();
402         (*tri)->point2 = new Coord();
403         (*tri)->point3 = new Coord();
404 
405         result = hMesh2D.FindTri( point->xVal, point->yVal, point->zVal, *tri);
406 
407         return result;
408     }
409 
410     template<class NODE>
411     bool World<NODE>::GetTri(Face_handle* tri, Point* point)
412     {
413         return hMesh2D.FindTri(point->x(), point->y(), point->z(), tri);
414     }
415 
416     /** Returns node for current location
417      */
418     template<class NODE>
419     NODE* World<NODE>::GetCurrentNode()
420     {
421         return currentLocNode;
422     }
```

```
423
424     /** Returns node for desired location to reach
425      */
426     template<class NODE>
427     NODE* World<NODE>::GetGoalNode()
428     {
429         return goalNode;
430     }
431
432     /** Take the given Coord pointer, create a new Coord with matching values,
            such that the one passed in can safely be freed/deleted by external
            functions
433      */
434     template<class NODE>
435     void World<NODE>::AddPoint(Coord *newPoint)
436     {
437         Triangle* encompTri = new Triangle;
438         if(GetTri(&encompTri, newPoint))
439         {
440             bool addTri = true;
441             int zValue = 0;
442
443             zValue = encompTri->InterpolateZValue(*newPoint);
444             if(abs(zValue - newPoint->z()) <= COPLANAR_PRECISION)
445             {
446                 float dist1 = CalculateDirectDistance(*newPoint, *encompTri->
                        point1, false);
447                 float dist2 = CalculateDirectDistance(*newPoint, *encompTri->
                        point2, false);
448                 float dist3 = CalculateDirectDistance(*newPoint, *encompTri->
                        point3, false);
449                 if(! (dist1 > COPLANAR_DIST_THRESHOLD && dist2 >
                        COPLANAR_DIST_THRESHOLD && dist3 > COPLANAR_DIST_THRESHOLD) )
450                     addTri = false;
451             }
452             else if(abs(zValue - newPoint->z()) <= MEASUREMENT_PRECISION)
453             {
454                 float dist1 = CalculateDirectDistance(*newPoint, *encompTri->
                        point1, false);
455                 float dist2 = CalculateDirectDistance(*newPoint, *encompTri->
                        point2, false);
456                 float dist3 = CalculateDirectDistance(*newPoint, *encompTri->
                        point3, false);
457                 if( !(dist1 > DIST_THRESHOLD && dist2 > DIST_THRESHOLD && dist3 >
                         DIST_THRESHOLD) )
458                     addTri = false;
459             }
```

```
460
461
462             if(addTri)
463                 hMesh2D.AddPoint(newPoint->x(), newPoint->y(), newPoint->z());
464         }
465         else //outside held area
466         {
467             float dist1 = CalculateDirectDistance(*newPoint, *encompTri->point1,
                    false);
468             float dist2 = CalculateDirectDistance(*newPoint, *encompTri->point2,
                    false);
469
470             if(dist1 > UNKNOWN_DIST_THRESHOLD && dist2 > UNKNOWN_DIST_THRESHOLD)
471                 hMesh2D.AddPoint(newPoint->x(), newPoint->y(), newPoint->z());
472         }
473
474         delete encompTri->point1;
475         delete encompTri->point2;
476         delete encompTri->point3;
477         encompTri->point1 = NULL;
478         encompTri->point2 = NULL;
479         encompTri->point3 = NULL;
480         delete encompTri;
481         encompTri = NULL;
482     }
483
484     template<class NODE>
485     void World<NODE>::AddHazard(Hazard hazard)
486     {
487         int numCoords = hazard.GetNumPoints();
488         Point* points = hazard.GetPoints();
489
490         if(SHOW_INFO) printf("Number of data points of hazard added %d \n",
                numCoords);
491
492         if(points != NULL && numCoords > 0)
493         {
494             for(int i = 0; i < numCoords; i++)
495             {
496                 AddPoint(&points[i]);
497             }
498
499             Face_handle* allTri = (Face_handle*) calloc(ARRAY_CHUNK, sizeof(
                    Face_handle));
500             int numTri = 0;
501
502             GetAllTriangles(&allTri, &numTri);
```

```
503             WriteTriToFile("mesh_triangles.txt", "wt", allTri, numTri, false);
504
505             FILE* output = fopen("/home/haz/NumTriInTessellation.txt", "at");
506             fprintf(output, "Number of triangles in this tessellation: %i\n",
                    numTri);
507             fclose(output);
508
509             free(allTri);
510             allTri = NULL;
511         }
512     }
513
514     template<class NODE>
515     void World<NODE>::DiscardPoints()
516     {
517         Face_handle* allTri = (Face_handle*) calloc(ARRAY_CHUNK, sizeof(
                Face_handle));
518         int numTri = 0;
519         GetAllTriangles(&allTri, &numTri);
520         int numPts = 0;
521         Point* outerPts = (Point*) calloc(numTri * 3, sizeof(Point));
522         for(int i = 0; i < numTri; i++)
523         {
524             Point pt1 = CalcTriCentre(*allTri[i]->GetPoint(0), *allTri[i]->
                    GetPoint(1), *allTri[i]->GetPoint(2));
525             if(CalculateDirectDistance(currentLocNode->GetPoint(), pt1, true) <
                    VISIBLE_DIST_MM * KEY_RATIO)
526             {
527                 allTri[i]->checked = true;
528                 continue;
529             }
530             else
531             {
532                 bool addPt0 = true, addPt1 = true, addPt2 = true;
533                 for(int j = 0; j < numPts; j++)
534                 {
535                     if(addPt0 && outerPts[j] == *allTri[i]->GetPoint(0))
536                         addPt0 = false;
537                     if(addPt1 && outerPts[j] == *allTri[i]->GetPoint(1))
538                         addPt1 = false;
539                     if(addPt2 && outerPts[j] == *allTri[i]->GetPoint(2))
540                         addPt2 = false;
541                 }
542
543                 if(addPt0)
544                     outerPts[numPts++] = *allTri[i]->GetPoint(0);
545                 if(addPt1)
```

```
546                     outerPts[numPts++] = *allTri[i]->GetPoint(1);
547                 if(addPt2)
548                     outerPts[numPts++] = *allTri[i]->GetPoint(2);
549             }
550
551 ///Following code is only used for hazard objects, and needs some work to
        complete it
552 //          if(allTri[i]->checked)
553 //              continue;
554 //          else
555 //              allTri[i]->checked = true;
556 //
557 //          if(allTri[i]->hazardType)
558 //          {
559 //              Hazard* newHazard = new Hazard();
560 //              newHazard->AddPoint(*allTri[i]->GetPoint(0));
561 //              newHazard->AddPoint(*allTri[i]->GetPoint(1));
562 //              newHazard->AddPoint(*allTri[i]->GetPoint(2));
563 //              savedHazards[numSavedHazards++] = newHazard;//simply add to
        terrains array of hazards
564 //              if(numSavedHazards%ARRAY_CHUNK == 0)
565 //              {
566 //                  void* tmp = realloc(savedHazards, (numSavedHazards +
        ARRAY_CHUNK) * sizeof(Hazard*));
567 //                  if(tmp != NULL)
568 //                      savedHazards = (Hazard**) tmp;
569 //              }
570 //
571 //              bool bounds = RecursExpandHazardObject(newHazard, allTri[i]);
572 //              bool joined = false;
573 //              int adjoiningHazard = 0;
574 //              for(int i = 0; i < numBorderHazards; i++)
575 //              {
576 //                  bool matches = newHazard->CompareHazard(borderHazards[i]);
577 //                  //see if hazard joins up with prev hazard which bordered edge
         of retained area
578 //                  //might join multiple ones
579 //
580 //                  if(matches)
581 //                  {
582 //                      if(joined)
583 //                          borderHazards[adjoiningHazard]->AdjoinHazard(
        borderHazards[i]);
584 //                      else
585 //                      {
586 //                          joined = true;
587 //                          adjoiningHazard = i;
```

```
588 //                              borderHazards[i]->AdjoinHazard(newHazard);
589 //                          }
590 //                      }
591 //                  }
592 //
593 //                  if(joined)
594 //                  {
595 //                      delete newHazard;
596 //                      if(!bounds)
597 //                      {
598 //                          borderHazards[adjoiningHazard] = NULL;
599 //                          //Call function to compact borderHazards - Otherwise can
        have seg faults when NULL is encountered inside array
600 //                      }
601 //                  }
602 //                  else if(bounds)
603 //                  {
604 //                      borderHazards[numBorderHazards++] = newHazard;
605 //                      if(numBorderHazards%LIST_CHUNK == 0)
606 //                      {
607 //                          void* tmp = realloc(borderHazards, (numBorderHazards +
        LIST_CHUNK) * sizeof(Hazard*));
608 //                          if(tmp != NULL)
609 //                              borderHazards = (Hazard**) tmp;
610 //                      }
611 //                  }
612 //              }
613         }
614
615         for(int i = 0; i < numTri; i++)
616             allTri[i]->checked = false;
617
618         for(int i = 0; i < numPts; i++)
619         {
620             hMesh2D.RemoveVert(outerPts[i]);
621         }
622
623         free(allTri);
624         free(outerPts);
625     }
626
627     /** Recursively search around a hazard to find neighbours which join it to
            form a hazard
628     */
629     template<class NODE>
630     bool World<NODE>::RecursExpandHazardObject(Hazard *newHazard, Face_handle
            hazTri)
```

```
631     {
632         bool boundsHeldArea = false;
633         for(int i = 0; i < 3; i++)
634         {
635             if(hazTri->neighbor(i)->checked)
636                 continue;
637             else
638                 hazTri->neighbor(i)->checked = true;
639 
640             if(hazTri->neighbor(i)->hazardType && !hMesh2D.CheckIfInfinite<
                    Face_handle>(hazTri->neighbor(i)))
641             {
642                 Point pt1 = CalcTriCentre(*hazTri->neighbor(i)->GetPoint(0), *
                        hazTri->neighbor(i)->GetPoint(1), *hazTri->neighbor(i)->
                        GetPoint(2));
643                 if(CalculateDirectDistance(currentLocNode->GetPoint(), pt1, true)
                        < VISIBLE_DIST_MM * KEY_RATIO)
644                     boundsHeldArea = true;//issue as hazard bounds radius of
                            being kept
645                 else
646                 {
647                     newHazard->AddPoint(*hazTri->neighbor(i)->GetPoint(0));
648                     newHazard->AddPoint(*hazTri->neighbor(i)->GetPoint(1));
649                     newHazard->AddPoint(*hazTri->neighbor(i)->GetPoint(2));
650                     boundsHeldArea = RecursExpandHazardObject(newHazard, hazTri->
                            neighbor(i));
651                 }
652             }
653         }
654 
655         return boundsHeldArea;
656     }
657 
658     template<class NODE>
659     void World<NODE>::GetAllTriangles(Triangle*** allTriangles, int *numTri)
660     {
661         hMesh2D.GetTrianglesIter(allTriangles, numTri);
662 
663         return;
664     }
665 
666     template<class NODE>
667     void World<NODE>::GetAllTriangles(Face_handle** allTriangles, int *numTri)
668     {
669         hMesh2D.GetTrianglesIter(allTriangles, numTri);
670 
671         return;
```

```
672     }
673
674     /** Kind of superfluous to have a function for this, as the parameters are
            visible from robot, so could be set at creation.
675     */
676     template<class NODE>
677     void World<NODE>::SetHazardIdentParam(int sAngle, int wAngle, int rLength,
            int rWidth, int aSep, int wRadius, int wWidth, int cBuffer, float mDist,
            float vDist)
678     {
679         SLOPEANGLE = sAngle;
680         WALLANGLE = wAngle;
681         ROBOTLENGTH = rLength;
682         ROBOTWIDTH = rWidth;
683         AXLESEP = aSep;
684         WHEELRADIUS = wRadius;
685         WHEELWIDTH = wWidth;
686         CLEARANCEBUFFER = cBuffer;
687         MOVE_INCR = mDist;
688         VISIBLE_DIST_MM = vDist;
689     }
690
691     template<class NODE>
692     void World<NODE>::LoadKnown(bool nodesKnown)
693     {
694         char buffer[90];
695         FILE* hazardsPath = fopen("../known_hazards.txt", "rt");
696         if(hazardsPath == NULL)
697         {
698             if(SHOW_ERRORS) printf("Error - Unable to open file containing
                    bounding boxes of known hazards\n");
699             return;
700         }
701
702         int x1,x2,y1,y2;
703         numPreMadeHazards = 0;
704         preKnownHazards = (Point*) calloc(ARRAY_CHUNK, sizeof(Point));
705         while(fgets(buffer, 90, hazardsPath) != NULL)
706         {
707             if(buffer[0] != '\n')
708             {
709                 if(numPreMadeHazards != 0 && numPreMadeHazards%ARRAY_CHUNK == 0)
710                 {
711                     void* tmp = realloc(preKnownHazards, (numPreMadeHazards +
                            ARRAY_CHUNK) * sizeof(Point));
712                     if(tmp != NULL)
713                         preKnownHazards = (Point*) tmp;
```

```
714                 }
715
716                 sscanf(buffer, "%d %d %d %d", &x1, &y1, &x2, &y2);
717                 preKnownHazards[numPreMadeHazards++] = Point(x1 * SCALE_FACTOR,
                        y1 * SCALE_FACTOR, 0);
718                 preKnownHazards[numPreMadeHazards++] = Point(x2 * SCALE_FACTOR,
                        y2 * SCALE_FACTOR, 0);
719             }
720         }
721
722         if(nodesKnown)
723         {
724             FILE* pointsPath = fopen("../known_points.txt", "rt");
725             if(pointsPath == NULL)
726             {
727                 if(SHOW_ERRORS) printf("Error - Unable to open file containing
                        pre-generated way points\n");
728                 return;
729             }
730
731             int x,y;
732             numPreMadePoints = 0;
733             preMadePoints = (Point*) calloc(ARRAY_CHUNK, sizeof(Point));
734             preMadeAdded = (bool*) calloc(ARRAY_CHUNK, sizeof(bool));
735             while(fgets(buffer, 90, pointsPath) != NULL)
736             {
737                 if(buffer[0] != '\n')
738                 {
739                     if(numPreMadePoints != 0 && numPreMadePoints%ARRAY_CHUNK ==
                            0)
740                     {
741                         void* tmp = realloc(preMadePoints, (numPreMadePoints +
                                ARRAY_CHUNK) * sizeof(Point));
742                         if(tmp != NULL)
743                             preMadePoints = (Point*) tmp;
744
745                         tmp = realloc(preMadeAdded, (numPreMadePoints +
                                ARRAY_CHUNK) * sizeof(bool));
746                         if(tmp != NULL)
747                             preMadeAdded = (bool*) tmp;
748                     }
749
750                     sscanf(buffer, "%d %d", &x, &y);
751                     preMadeAdded[numPreMadePoints] = false;
752                     preMadePoints[numPreMadePoints++] = Point(x * SCALE_FACTOR, y
                            * SCALE_FACTOR, 0);
753                 }
```

```
754             }
755         }
756     }
757
758     /** Calls the hazard identification functions
759      */
760     template<class NODE>
761     void World<NODE>::ProcessPoints(bool known)
762     {
763         if(known)
764             IdentifyKnownHazards<Face_handle, Point>(0, NULL);
765         else
766             IdentifyHazards<Face_handle, Point>(0, NULL);
767     }
768
769     template<class NODE>
770     void World<NODE>::GenHazardNodes(HazardNode** hNodes, int* numNodes, Point
            dirVect)
771     {
772         Point normVect(0,0,0), frontLeft(0,0,0), frontRight(0,0,0);
773         Face_handle* allTriangles = (Face_handle*) calloc(ARRAY_CHUNK, sizeof(
                Face_handle));
774         int numTri = 0;
775         GetAllTriangles(&allTriangles, &numTri);
776
777         if(graphInitialised)
778         {
779             //dirVect initially scaled to 10, to avoid overflow in cross product,
                    then it and normVect are scaled to visible dist
780             dirVect =  NormaliseVector(dirVect, 10, false);
781             normVect = ScaledCrossProduct(dirVect, dirVect + (Point(0,0,10) -
                    ZERO_POINT), 10);
782             frontLeft = dirVect - (normVect - ZERO_POINT);
783             frontRight = dirVect + (normVect - ZERO_POINT);
784             ///Alter so is norm vectors ie, just gets forward half rather than
                    central quarter?
785         }
786
787         WriteLineToFile("links.txt", "wt", (Point**) NULL, 0);
788         WriteTriToFile<Face_handle>("hazards.txt", "wt", NULL, 0, true);
789
790         for(int i = 0; i < numTri; i++)
791         {
792             if(allTriangles[i]->hazardType)
793             {
794                 Face_handle temp = allTriangles[i];
795                 WriteTriToFile<Face_handle>("hazards.txt", "at", &temp, 1, true);
```

```
796                 for(int j = 0; j < 3; j++)
797                 {
798                     if(!allTriangles[i]->neighbor(j)->hazardType && !hMesh2D.
                            CheckIfInfinite<Face_handle>(allTriangles[i]->neighbor(j)
                            ))
799                     {
800                         Point *pt1, *pt2, midPt;
801                         pt1 = allTriangles[i]->GetPoint((j+1)%3);
802                         pt2 = allTriangles[i]->GetPoint((j+2)%3);
803 
804                         if(!graphInitialised || (ToTheLeftOfVector(frontRight,
                                currentLocNode->GetPoint(), *pt1) &&
805                         !ToTheLeftOfVector(frontLeft, currentLocNode->GetPoint(),
                                *pt1) &&
806                         CalculateDirectDistance(currentLocNode->GetPoint(), *pt1,
                                true) < VISIBLE_DIST_MM) ||
807                         (ToTheLeftOfVector(frontRight, currentLocNode->GetPoint()
                                , *pt2) &&
808                          !ToTheLeftOfVector(frontLeft, currentLocNode->GetPoint()
                                , *pt2) &&
809                         CalculateDirectDistance(currentLocNode->GetPoint(), *pt2,
                                true) < VISIBLE_DIST_MM))
810                         {
811                             midPt = *pt1 + (*pt2 - *pt1)/2;
812                             Point* edge[2] = {pt1, pt2};
813                             WriteLineToFile("links.txt", "at", edge, 2);
814 
815                             if(*numNodes%ARRAY_CHUNK == 0 && *numNodes != 0)
816                             {
817                                 void* tmp = realloc(*hNodes, (*numNodes +
                                        ARRAY_CHUNK) * sizeof(HazardNode));
818                                 if(tmp != NULL)
819                                     *hNodes = (HazardNode*) tmp;
820                             }
821 
822                             (*hNodes)[(*numNodes)++] = HazardNode(midPt,
                                    allTriangles[i], (j+1)%3, (j+2)%3);
823                         }
824                     }
825                 }
826             }
827         }
828 
829 
830         bool showCloseUpOfHazards = false;
831         if(!showCloseUpOfHazards)
832         {
```

```
833             Triangle* temp = new Triangle(&destination, &(destination + Coord
                    (2,2,0)), &(destination + Coord(-2,2,0)));
834             WriteTriToFile<Triangle*>("hazards.txt", "at", &temp, 1, true);
835             delete temp;
836
837             temp = new Triangle(&ConvertPointType<Point, Coord>(startPt), &(
                    ConvertPointType<Point, Coord>(startPt) + Coord(2,2,0)), &(
                    ConvertPointType<Point, Coord>(startPt) + Coord(-2,2,0)));
838             WriteTriToFile<Triangle*>("hazards.txt", "at", &temp, 1, true);
839             delete temp;
840         }
841
842         for(int k = 0, insertIndex = 0; k < *numNodes; k++)
843         {
844             if((*hNodes)[k].CheckValid())
845             {
846                 for(int l = k + 1; l < *numNodes; l++)
847                 {
848                     /// what if 1 in a million occurences that hazards points
                            happen to be generated vertical above each other ie at a
                            cliff?
849                     if((*hNodes)[l].CheckValid() && (*hNodes)[k] == (*hNodes)[l].
                            GetPoint())//is the point check enough, should
                            face_handle be tested too ie have hazardNode version of
                            oper==
850                     {
851                         (*hNodes)[l].vertA = -1;
852                         (*hNodes)[l].vertB = -1;
853                         if(SHOW_ERRORS) printf("Error - duplicate hazardNodes
                                were found\n");
854                     }
855                 }
856                 (*hNodes)[insertIndex++] = (*hNodes)[k];//nullify l or shift all
857             }
858         }
859
860         free(allTriangles);
861     }
862
863     /** Generate MapNodes from the given HazardNodes
864         If no hazards, then just add start & goal
865         While hNodes and mNodes must be within visi dist, keyNodes are within
                1.41 * visi dist
866     */
867     template<class NODE>
868     void World<NODE>::GenMapNodes(HazardNode* hNodes, int numNodes, Point dirVect
            )
```

```
869     {
870         Point normVect(0,0,0), frontLeft(0,0,0), frontRight(0,0,0), currentPos
                (0,0,0);
871         NODE** mNodes = (NODE**) calloc(ARRAY_CHUNK, sizeof(NODE*));
872         int numMapNodes = 0;
873         bool isSlope = false;
874 
875         if(numNodes == 0)
876             return;
877 
878         //perhaps want to use fLeft and fRight even if not initiliased, otherwise
                first loop will likely get no mapNodes
879         if(graphInitialised)
880         {
881             if(dirVect.x() == 0 && dirVect.y() == 0 && dirVect.z() == 0)
882                 return;
883 
884             dirVect =  NormaliseVector(dirVect, 10, false);
885             normVect = ScaledCrossProduct(dirVect, dirVect + (Point(0,0,10) -
                    ZERO_POINT), VISIBLE_DIST_MM);
886             dirVect =  NormaliseVector(dirVect, VISIBLE_DIST_MM, false);
887 
888             if(dirVect.x() == 0 && dirVect.y() == 0 && dirVect.z() == 0)
889                 return;
890 
891             currentPos = currentLocNode->GetPoint();
892             frontLeft = currentPos + /*KEY_RATIO*/ (dirVect - normVect);
893             frontRight = currentPos + /*KEY_RATIO*/ ((dirVect - ZERO_POINT) + (
                    normVect - ZERO_POINT));
894 
895             if(CheckBetweenHazardNodes(frontLeft, frontRight, &isSlope))
896             {
897                 if(numMapNodes%ARRAY_CHUNK == 0 && numMapNodes != 0)
898                 {
899                     void* tmp = realloc(mNodes, (numMapNodes + ARRAY_CHUNK) *
                            sizeof(NODE*));
900                     if(tmp != NULL)
901                         mNodes = (NODE**) tmp;
902                 }
903 
904                 mNodes[numMapNodes++] = new NODE(frontLeft + (frontRight -
                        frontLeft)/2, (Face_handle*) NULL, (Face_handle*) NULL);
905             }
906         }
907 
908         for(int i = 0; i < numNodes; i++)
909         {
```

```
910             int startNumMapNodes = numMapNodes;
911
912             Point pt1(0,0,0), pt1A(0,0,0), pt1B(0,0,0), pt1C(0,0,0);
913             bool nullTri1 = false;
914             pt1 = hNodes[i].GetPoint();
915             if(hNodes[i].GetHazardTri() != NULL)
916             {
917                 pt1A = *hNodes[i].GetHazardTri()->GetPoint(hNodes[i].vertA);
918                 pt1B = *hNodes[i].GetHazardTri()->GetPoint(hNodes[i].vertB);
919                 pt1C = *hNodes[i].GetHazardTri()->GetPoint(3 - (hNodes[i].vertB +
                        hNodes[i].vertA));
920             }
921             else
922                 nullTri1 = true;
923
924
925             for(int j = i + 1; j < numNodes; j++)
926             {
927                 //check relative position/sides
928                 Point  pt2(0,0,0), pt2A(0,0,0), pt2B(0,0,0), pt2C(0,0,0);
929                 pt2 = hNodes[j].GetPoint();
930                 Point midPoint = pt1 + (pt2 - pt1)/2;
931                 int angle1 = 90, angle2 = 90;
932                 int res = InsideTriangleTest(midPoint, currentPos, frontLeft,
                        frontRight);
933                 if(graphInitialised && !res)
934                     continue;
935
936                 //if > 0 then pt2 is same side, else if pt1 is same side
937                 if(!nullTri1)
938                 {
939                     int val = ((pt1B.x() - pt1A.x()) * (pt2.y() - pt1A.y()) - (
                            pt2.x() - pt1A.x()) * (pt1B.y() - pt1A.y())) * ((pt1B.x()
                             - pt1A.x()) * (pt1C.y() - pt1A.y()) - (pt1C.x() - pt1A.x
                            ()) * (pt1B.y() - pt1A.y()));
940
941                     if(((pt1B.x() - pt1A.x()) * (pt2.y() - pt1A.y()) - (pt2.x() -
                             pt1A.x()) * (pt1B.y() - pt1A.y())) * ((pt1B.x() - pt1A.x
                            ()) * (pt1C.y() - pt1A.y()) - (pt1C.x() - pt1A.x()) * (
                            pt1B.y() - pt1A.y())) >= 0)
942                         continue;
943                     else
944                         angle1 = AngleBetweenTwoVectors(pt2 - pt1, pt1B - pt1A);
945                 }
946
947                 if(hNodes[j].GetHazardTri() != NULL)
948                 {
```

```
949                     pt2A = *hNodes[j].GetHazardTri()->GetPoint(hNodes[j].vertA);
950                     pt2B = *hNodes[j].GetHazardTri()->GetPoint(hNodes[j].vertB);
951                     pt2C = *hNodes[j].GetHazardTri()->GetPoint(3 - (hNodes[j].
                            vertB + hNodes[j].vertA));
952                     if(((pt2B.x() - pt2A.x()) * (pt1.y() - pt2A.y()) - (pt1.x() -
                             pt2A.x()) * (pt2B.y() - pt2A.y())) * ((pt2B.x() - pt2A.x
                            ()) * (pt2C.y() - pt2A.y()) - (pt2C.x() - pt2A.x()) * (
                            pt2B.y() - pt2A.y())) >= 0)
953                         continue;
954                     else
955                         angle2 = AngleBetweenTwoVectors(pt2 - pt1, pt2B - pt2A);
                                //should this be reversed to pt1 - pt2 ?
956                 }
957
958                 int dist = CalculateDirectDistance(pt1, pt2, false);
959                 //possibly have tested in conjunction with dist between points,
                        as low incidence ok if very large dist
960                 if(dist < CLEARANCEBUFFER || angle1 <= MIN_INCIDENT_ANGLE ||
                        angle1 >= 180 - MIN_INCIDENT_ANGLE || angle2 <=
                        MIN_INCIDENT_ANGLE || angle2 >= 180 - MIN_INCIDENT_ANGLE)
961                     continue;
962                 else
963                 {
964                     isSlope = false;
965                     if(CheckBetweenHazardNodes(pt1, pt2, &isSlope))
966                     {
967                         if(numMapNodes%ARRAY_CHUNK == 0 && numMapNodes != 0)
968                         {
969                             void* tmp = realloc(mNodes, (numMapNodes +
                                    ARRAY_CHUNK) * sizeof(NODE*));
970                             if(tmp != NULL)
971                                 mNodes = (NODE**) tmp;
972                         }
973
974                         mNodes[numMapNodes++] = new NODE(pt1 + (pt2 - pt1)/2,
                                hNodes[i].GetHazardTri(), hNodes[j].GetHazardTri());
975                     }
976                     else if(isSlope)
977                     {
978                         //TODO Uncomment slopes and have one way links utilised
979 //                          if((numMapNodes%ARRAY_CHUNK == 0 || (numMapNodes + 1)%
    ARRAY_CHUNK == 0) && numMapNodes != 0)
980 //                          {
981 //                              void* tmp = realloc(mNodes, (numMapNodes +
    numMapNodes%2 + ARRAY_CHUNK) * sizeof(NODE*));
982 //                              if(tmp != NULL)
983 //                                  mNodes = (NODE**) tmp;
```

```
984 //                      }
985 //                      //Have it that if hazA and hazB are same, indicates a
        slope-slope link point
986 //                      mNodes[numMapNodes++] = new NODE(pt1, hNodes[i].
        GetHazardTri(), hNodes[i].GetHazardTri());//have hazardTri added, so validity
         of this node can be checked later?
987 //                      mNodes[numMapNodes++] = new NODE(pt2, hNodes[j].
        GetHazardTri(), hNodes[j].GetHazardTri());//have hazardTri added, so validity
         of this node can be checked later?
988 //                      mNodes[numMapNodes - 2]->AddLink(mNodes[numMapNodes - 1])
        ;
989 //                      mNodes[numMapNodes - 1]->AddLink(mNodes[numMapNodes - 2])
        ;
990                     }
991                 }
992             }
993
994             if(graphInitialised && numMapNodes - startNumMapNodes <= 2)//less
                than how much?
995             {
996                 //try linking to frontLeft, frontRight
997                 isSlope = false;
998                 if(CheckBetweenHazardNodes(pt1, frontLeft, &isSlope))
999                 {
1000                    if(numMapNodes%ARRAY_CHUNK == 0 && numMapNodes != 0)
1001                    {
1002                        void* tmp = realloc(mNodes, (numMapNodes + ARRAY_CHUNK) *
                            sizeof(NODE*));
1003                        if(tmp != NULL)
1004                            mNodes = (NODE**) tmp;
1005                    }
1006
1007                    mNodes[numMapNodes++] = new NODE(pt1 + (frontLeft - pt1)/2,
                        hNodes[i].GetHazardTri(), NULL);
1008                }
1009
1010                if(CheckBetweenHazardNodes(pt1, frontRight, &isSlope))
1011                {
1012                    if(numMapNodes%ARRAY_CHUNK == 0 && numMapNodes != 0)
1013                    {
1014                        void* tmp = realloc(mNodes, (numMapNodes + ARRAY_CHUNK) *
                            sizeof(NODE*));
1015                        if(tmp != NULL)
1016                            mNodes = (NODE**) tmp;
1017                    }
1018
```

```
1019                    mNodes[numMapNodes++] = new NODE(pt1 + (frontRight - pt1)/2,
                            hNodes[i].GetHazardTri(), NULL);
1020                }
1021            }
1022        }
1023
1024        if(!graphInitialised)
1025            graphInitialised = true;
1026
1027        if(numMapNodes > 0)
1028        {
1029            for(int k = 0; k < numMapNodes; k++)
1030            {
1031                for(int l = k + 1; l < numMapNodes; l++)
1032                {
1033                    if(XYMatch(mNodes[k]->GetPoint(), mNodes[l]->GetPoint()))
1034                    {
1035                        //if(mNodes[l]->hazardType == SLOPE)
1036                        //addlink to mNodes[k] of the other slope which [l] links
                                to
1037                        mNodes[l]->MarkNode();
1038                    }
1039                }
1040            }
1041
1042            CullKeyNodes(mNodes, &numMapNodes, dirVect);//keep those within
                radius of 1.41 * view dist but discard ones which conflict with
                seen info
1043            DiscardPoints();
1044
1045
1046            LinkMapNodes(mNodes, &numMapNodes);//within viewed area
1047            LinkKeyNodes(mNodes, &numMapNodes, dirVect);//within radius of 1.41 *
                 view dist
1048            LinkOtherNodes();
1049
1050
1051            for(int i = 0; i < numMapNodes; i++)
1052            {
1053                if(mNodes[i]->CheckForMark())
1054                {
1055                    //heldNodes will do the freeing otherwise it doesn't know and
                         tries to test freed nodes
1056                    ///what if hasn't been added to heldNodes yet? could a new
                        node be marked
1057    //              delete mNodes[i];
1058                    mNodes[i] = NULL;
```

```
1059                }
1060                else
1061                {
1062                    if(numHeldNodes%ARRAY_CHUNK == 0)
1063                    {
1064                        void* tmp = realloc(heldNodes, (numHeldNodes +
                                ARRAY_CHUNK) * sizeof(NODE*));
1065                        if(tmp != NULL)
1066                            heldNodes = (NODE**) tmp;
1067                    }
1068
1069                    heldNodes[numHeldNodes++] = mNodes[i];
1070                }
1071            }
1072        }
1073
1074        free(mNodes);
1075        mNodes = NULL;
1076    }
1077
1078    /** Generate set of MapNodes given position and premade node
1079    */
1080    template<class NODE>
1081    void World<NODE>::GenPreMadeMapNodes(Point dirVect)
1082    {
1083        NODE** mNodes = (NODE**) calloc(ARRAY_CHUNK, sizeof(NODE*));
1084        int numMapNodes = 0;
1085        Point currentPos = currentLocNode->GetPoint();
1086
1087        for(int i = 0; i < numPreMadePoints; i++)
1088        {
1089            if(/*!preMadeAdded[i] &&*/ CalculateDirectDistance(preMadePoints[i],
                    currentPos, true) < VISIBLE_DIST_MM)
1090            {
1091                Point dirVect = ZERO_POINT + (preMadePoints[i] - currentPos);
1092                Point normVect = CrossProduct(dirVect, dirVect + (Point(0,0,10) -
                        ZERO_POINT));
1093                normVect = NormaliseVector<Point>(normVect , ROBOTWIDTH/2, false)
                        ;
1094
1095                if(hMesh2D.CheckLineForHazards(currentPos, preMadePoints[i],
                        false, normVect, false, NULL))
1096                {
1097                    if(numMapNodes%ARRAY_CHUNK == 0 && numMapNodes != 0)
1098                    {
1099                        void* tmp = realloc(mNodes, (numMapNodes + ARRAY_CHUNK) *
                                sizeof(NODE*));
```

```
1100                        if(tmp != NULL)
1101                            mNodes = (NODE**) tmp;
1102                    }
1103
1104                    mNodes[numMapNodes++] = new NODE(preMadePoints[i], NULL, NULL
                            );
1105                    preMadeAdded[i] = true;
1106                }
1107            }
1108        }
1109
1110        if(!graphInitialised)
1111            graphInitialised = true;
1112
1113        //Nodes get added to keyNodes at end of LinkKeyNodes, when they merge.
1114        //Culled if within visi_range and not a matching mapNode. Hence skipping
              nodes already added is leading to loss of links
1115        if(numMapNodes > 0)
1116        {
1117            for(int k = 0; k < numMapNodes; k++)
1118            {
1119                for(int l = k + 1; l < numMapNodes; l++)
1120                {
1121                    if(XYMatch(mNodes[k]->GetPoint(), mNodes[l]->GetPoint()))
1122                        mNodes[l]->MarkNode();
1123                }
1124            }
1125
1126            CullKeyNodes(mNodes, &numMapNodes, dirVect);//keep those within
                  radius of 1.41 * view dist but discard ones which conflict with
                  seen info
1127            DiscardPoints();
1128
1129            LinkMapNodes(mNodes, &numMapNodes);//within viewed area
1130            LinkKeyNodes(mNodes, &numMapNodes, dirVect);//within radius of 1.41 *
                   view dist
1131        }
1132
1133        LinkOtherNodes();
1134
1135        for(int i = 0; i < numMapNodes; i++)
1136        {
1137            if(mNodes[i]->CheckForMark())
1138                mNodes[i] = NULL;
1139            else
1140            {
1141                if(numHeldNodes%ARRAY_CHUNK == 0)
```

```
1142                {
1143                    void* tmp = realloc(heldNodes, (numHeldNodes + ARRAY_CHUNK) *
                            sizeof(NODE*));
1144                    if(tmp != NULL)
1145                        heldNodes = (NODE**) tmp;
1146                }
1147
1148                heldNodes[numHeldNodes++] = mNodes[i];
1149            }
1150        }
1151
1152        free(mNodes);
1153        mNodes = NULL;
1154    }
1155
1156    template<class NODE>
1157    void World<NODE>::CullHeldNodes()
1158    {
1159        CullNodeSets();
1160
1161        int heldNodeIndex = 0;
1162        for(int i = 1; i < numHeldNodes; i++)
1163        {
1164            if(heldNodes[i] == NULL)
1165                continue;
1166            else if(heldNodes[i]->CheckForMark())
1167            {
1168                delete heldNodes[i];
1169                heldNodes[i] = NULL;
1170            }
1171            else
1172                heldNodes[heldNodeIndex++] = heldNodes[i];
1173        }
1174
1175        if(heldNodeIndex != numHeldNodes)
1176        {
1177            numHeldNodes = heldNodeIndex;
1178            void* tmp = realloc(heldNodes, (numHeldNodes + ARRAY_CHUNK -
                    numHeldNodes%ARRAY_CHUNK) * sizeof(NODE*));
1179            if(tmp != NULL)
1180                heldNodes = (NODE**) tmp;
1181        }
1182    }
1183
1184    template<class NODE>
1185    void World<NODE>::CullNodeSets()
1186    {
```

```
1187            bool nullClosed = false;
1188            bool nullOpen = false;
1189
1190            for(int i = 0; i < closedInd; i++)
1191            {
1192                if(closedSet[i] != NULL && closedSet[i]->CheckForMark())
1193                {
1194                    closedSet[i] = NULL;
1195                }
1196            }
1197
1198            for(int i = 0; i < openInd; i++)
1199            {
1200                if(openSet[i] != NULL && openSet[i]->CheckForMark())
1201                {
1202                    openSet[i] = NULL;
1203                }
1204            }
1205        }
1206
1207        /**
1208            Cull keyNodes within visible area, Ignore if marked or outside area,
                otherwise only keep if is current location or a mNode matches with it
                 exactly in which case recheck links.
1209            Cull keyNodes which are outside key_ratio * visi_dist of current point
1210            Takes order(n) to remove culled points and readjust/shift values in array
1211            The value shifting make not be necessary, could just null out the
                pointers and have them ignore by other functions using keyNodes
1212        */
1213        template<class NODE>
1214        void World<NODE>::CullKeyNodes(NODE** mNodes, int* numMapNodes, Point dirVect
            )
1215        {
1216            Point normVect, frontLeft, frontRight, currentPos;
1217            //dirVect initially scaled to 10, to avoid overflow in cross product,
                then it and normVect are scaled to visible dist
1218            dirVect =  NormaliseVector(dirVect, 10, false);
1219            normVect = ScaledCrossProduct(dirVect, dirVect + (Point(0,0,10) -
                ZERO_POINT), VISIBLE_DIST_MM);
1220            dirVect =  NormaliseVector(dirVect, VISIBLE_DIST_MM, false);
1221            currentPos = currentLocNode->GetPoint();
1222            frontLeft = currentPos + KEY_RATIO * (dirVect - normVect);
1223            frontRight = currentPos + KEY_RATIO * ((dirVect - ZERO_POINT) + (normVect
                 - ZERO_POINT));
1224
1225            int insertIndex = 0;
1226            for(int i = 0; i < numKeyNodes; i++)
```

```
1227        {
1228            if(keyNodes[i]->CheckForMark())
1229                continue;
1230
1231            float dist = CalculateDirectDistance(keyNodes[i]->GetPoint(),
                    currentLocNode->GetPoint(), true);
1232            if(dist < VISIBLE_DIST_MM * KEY_RATIO)
1233            {
1234                if(keyNodes[i] == goalNode)
1235                    keyNodes[insertIndex++] = keyNodes[i];
1236                else if(InsideTriangleTest(keyNodes[i]->GetPoint(), currentPos,
                        frontLeft, frontRight))
1237                {
1238                    int j = 0;
1239                    for( j = 0; j < *numMapNodes; j++)
1240                    {
1241                        if(mNodes[j]->CheckForMark())
1242                            continue;
1243
1244                        Point pt1 = keyNodes[i]->GetPoint(), pt2 = mNodes[j]->
                            GetPoint();
1245
1246                        if(XYMatch(pt1,pt2))
1247                        {
1248                            mNodes[j]->MarkNode();
1249                            keyNodes[insertIndex++] = keyNodes[i];
1250                            break;//issue if more than one mNode with same point,
                                 though that should have been tested when gen
                                hNodes
1251                        }
1252                        else if(CalculateDirectDistance(pt1, pt2, true) <
                            MAPNODE_PROXIMITY_THRESHOLD)
1253                        {
1254                            float tDist = CalculateDirectDistance(pt1, pt2, true)
                                ;
1255                            //as mNodes are withn prox_thresh of each other can
                                deduce, is no matching mNode for pt1
1256                            keyNodes[i]->MarkNode();
1257                            keyNodes[i] = NULL;
1258                            break;
1259                        }
1260                    }
1261
1262                    //if doesn't match an mNode and not within prox_dist of
                        another one, keep or discard?
1263                    if(j >= *numMapNodes)
1264                    {
```

```
1265 //                      keyNodes[insertIndex++] = keyNodes[i];
1266                         keyNodes[i]->MarkNode();
1267                         keyNodes[i] = NULL;
1268                     }
1269                 }
1270                 else if(dist > VISIBLE_DIST_MM) //is in outer edge of area, so
                         shouldn't have had it's triangles changed?? what about due to
                          the cull?
1271                     keyNodes[insertIndex++] = keyNodes[i];
1272                 else // check if triangles have changed?
1273                 {
1274                     Face_handle faceA, faceB, test;
1275                     Point centreA, centreB;
1276                     faceA = keyNodes[i]->GetHazardTriA();
1277                     faceB = keyNodes[i]->GetHazardTriB();
1278 
1279                     if(faceA == NULL || faceB == NULL)
1280                     {
1281                         keyNodes[i]->MarkNode();
1282                         keyNodes[i] = NULL;
1283                     }
1284                     else if(!faceA->CheckValidTri() || !faceB->CheckValidTri())
1285                     {
1286                         keyNodes[i]->MarkNode();
1287                         keyNodes[i] = NULL;
1288                     }
1289                     else if(!faceA->hazardType || !faceB->hazardType)
1290                     {
1291                         keyNodes[i]->MarkNode();
1292                         keyNodes[i] = NULL;
1293                     }
1294                     else
1295                     {
1296                         centreA = CalcTriCentre(*faceA->GetPoint(0), *faceA->
                                 GetPoint(1), *faceA->GetPoint(2));
1297                         centreB = CalcTriCentre(*faceB->GetPoint(0), *faceB->
                                 GetPoint(1), *faceB->GetPoint(2));
1298 
1299                         if( !(GetTri(&test, &centreA) && test == faceA && test->
                                 hazardType) || !(GetTri(&test, &centreB) && test ==
                                 faceB && test->hazardType))
1300                         {
1301                             keyNodes[i]->MarkNode();
1302                             keyNodes[i] = NULL;
1303                         }
1304                         else
1305                             keyNodes[insertIndex++] = keyNodes[i];
```

```
1306                    }
1307                }
1308            }
1309        }
1310
1311        for(int i = 0; i < insertIndex; i++)
1312        {
1313            if(keyNodes[i]->CheckForMark())
1314                continue;//shouldn't trigger
1315
1316            float dist = CalculateDirectDistance(keyNodes[i]->GetPoint(),
                    currentLocNode->GetPoint(), true);
1317            if(dist < VISIBLE_DIST_MM && InsideTriangleTest(keyNodes[i]->GetPoint
                    (), currentPos, frontLeft, frontRight))
1318            {
1319                Point pt1 = keyNodes[i]->GetPoint();
1320                int numInitLinks = keyNodes[i]->GetNumLinks();
1321                for(int j = 0; j < keyNodes[i]->GetNumLinks(); )//numLinks will
                        change as points unlinked
1322                {
1323                    Point pt2 = keyNodes[i]->FollowLink(j)->GetPoint();
1324                    Point dirVect = ZERO_POINT + (pt2 - pt1);
1325                    Point normVect = CrossProduct(dirVect, dirVect + (Point
                            (0,0,10) - ZERO_POINT));
1326                    if(CheckBetweenPoints(pt1, pt2, true, normVect))
1327                    {
1328                        keyNodes[i]->FollowLink(j)->BreakLink(keyNodes[i]);
1329                        keyNodes[i]->BreakLink(keyNodes[i]->FollowLink(j));
1330                    }
1331                    else
1332                        j++;
1333                }
1334            }
1335        }
1336
1337        if(insertIndex != numKeyNodes)
1338        {
1339            numKeyNodes = insertIndex;
1340            void* tmp = realloc(keyNodes, (numKeyNodes + ARRAY_CHUNK -
                    numKeyNodes%ARRAY_CHUNK) * sizeof(NODE*));
1341            if(tmp != NULL)
1342                keyNodes = (NODE**) tmp;
1343        }
1344    }
1345
1346    /** link MapNodes. Ignore/remove mapNodes too close together to cut down on
            number of links as can end up with many superfluous ones. Link nodes with
```

```
        clear paths between them and enough clearance on sides.
1347    */
1348    template<class NODE>
1349    void World<NODE>::LinkMapNodes(NODE** mNodes, int* numMapNodes){
1350        WriteColourChangeToFile("links.txt");
1351        int dIndex1 = 0, dIndex2 = 0, dIndex3 = 0;
1352        for(int i = 0; i < *numMapNodes; i++)
1353        {
1354            if(mNodes[i]->CheckForMark())
1355                continue;
1356
1357            for(int j = i + 1; j < *numMapNodes; j++)
1358            {
1359                if(mNodes[j]->CheckForMark())
1360                    continue;
1361                //if too close, ignore/remove -- should this be higher? what is
                        effect if is move distance?
1362                //if too far store and wait to see if have other links???
1363                //if one is slope and has a link already, then should keep that
                        one
1364                float dist1 = CalculateDirectDistance<Point>(mNodes[i]->GetPoint
                        (), mNodes[j]->GetPoint(), true);
1365
1366                if(dist1 < MAPNODE_PROXIMITY_THRESHOLD)
1367                    mNodes[j]->MarkNode();
1368            }
1369        }
1370
1371        for(int i = 0; i < *numMapNodes; i++)
1372        {
1373            if(mNodes[i]->CheckForMark())
1374                continue;
1375
1376            Point pt1 = mNodes[i]->GetPoint();
1377            for(int j = i + 1; j < *numMapNodes; j++)
1378            {
1379                if(mNodes[j]->CheckForMark())
1380                    continue;
1381
1382                float dist1 = CalculateDirectDistance<Point>(mNodes[i]->GetPoint
                        (), mNodes[j]->GetPoint(), true);
1383                if(dist1 > MAX_LINK_DISTANCE)
1384                    continue;
1385
1386                Point pt2 = mNodes[j]->GetPoint();
1387                Point dirVect = ZERO_POINT + (pt2 - pt1);
```

```
1388                Point normVect = CrossProduct(dirVect, dirVect + (Point(0,0,10) -
                        ZERO_POINT));
1389                normVect = NormaliseVector<Point>(normVect , ROBOTWIDTH/2, false)
                        ;
1390
1391                if(hMesh2D.CheckLineForHazards(pt1, pt2, true, normVect, false,
                        NULL))
1392                {
1393                    Point* edge[2] = {&pt1, &pt2};
1394                    WriteLineToFile("links.txt", "at", edge, 2);
1395                    mNodes[i]->AddLink(mNodes[j]);
1396                    mNodes[j]->AddLink(mNodes[i]);
1397                }
1398            }
1399        }
1400    }
1401
1402    /** link freshly created mNodes with keyNodes (and thus the whole graph).
        Links nodes with clear paths between them and enough clearance on sides.
        Also appends the mNodes to the keyNodes list
1403    */
1404    template<class NODE>
1405    void World<NODE>::LinkKeyNodes(NODE** mNodes, int* numMapNodes, Point dirVect
        )
1406    {
1407        int dIndex1 = 0, dIndex2 = 0, dIndex3 = 0;
1408        for(int i = 0; i < *numMapNodes; i++)
1409        {
1410            if(mNodes[i]->CheckForMark())
1411                continue;
1412
1413            Point pt1 = mNodes[i]->GetPoint();
1414            for(int j = 0; j < numKeyNodes; j++)
1415            {
1416                if(keyNodes[j]->CheckForMark())
1417                    continue;
1418
1419                Point pt2 = keyNodes[j]->GetPoint();
1420                if(pt1 == pt2)
1421                {
1422                }
1423                else if(XYMatch(pt1, pt2))
1424                {
1425                    CullKeyNodes(mNodes, numMapNodes, dirVect);
1426                }
1427                else
1428                {
```

```
1429                    float dist1 = CalculateDirectDistance<Point>(mNodes[i]->
                            GetPoint(), keyNodes[j]->GetPoint(), true);
1430                    if(dist1 > MAX_LINK_DISTANCE)
1431                        continue;
1432
1433                    Point dirVect = ZERO_POINT + (pt2 - pt1);
1434                    Point normVect = CrossProduct(dirVect, dirVect + (Point
                            (0,0,10) - ZERO_POINT));
1435                    Point tempNorm = normVect;
1436                    normVect = NormaliseVector<Point>(normVect , ROBOTWIDTH/2,
                            false);
1437                    //issue if pt1 and pt2 are the same. how does this occur -
                            mNodes created from crossing lines.?
1438
1439                    if(hMesh2D.CheckLineForHazards(pt1, pt2, true, normVect,
                            false, NULL))
1440                    {
1441                        Point* edge[2] = {&pt1, &pt2};
1442                        WriteLineToFile("links.txt", "at", edge, 2);
1443
1444                        mNodes[i]->AddLink((NODE*) keyNodes[j]);
1445                        keyNodes[j]->AddLink(mNodes[i]);
1446                    }
1447                }
1448            }
1449        }
1450
1451        MergeWithKeyNodes(mNodes, numMapNodes, false);
1452    }
1453
1454    template<class NODE>
1455    void World<NODE>::LinkOtherNodes()
1456    {
1457        bool goalIsKeyNode = false;
1458
1459        if(numKeyNodes > 0)
1460        {
1461            Point currentPt = currentLocNode->GetPoint();
1462            Point goalPt = goalNode->GetPoint();
1463            for(int i = 0; i < numKeyNodes; i++)
1464            {
1465                if(keyNodes[i]->CheckForMark())
1466                    continue;
1467
1468                Point pt2 = keyNodes[i]->GetPoint();
1469                Point dirVectCurr = ZERO_POINT + (pt2 - currentPt);
1470                Point dirVectGoal = ZERO_POINT + (pt2 - goalPt);
```

```
1471                Point normVectCurr = ScaledCrossProduct(dirVectCurr, dirVectCurr
                        + (Point(0,0,10) - ZERO_POINT), ROBOTWIDTH/2);
1472                Point normVectGoal = ScaledCrossProduct(dirVectGoal, dirVectGoal
                        + (Point(0,0,10) - ZERO_POINT), ROBOTWIDTH/2);
1473
1474                //have a test, so only try link if is within certain range, ie
                        only links once gets closer.
1475                if(goalPt == pt2)
1476                    goalIsKeyNode = true;
1477                else if(!XYMatch(goalPt, pt2) && CalculateDirectDistance(goalPt,
                        pt2, true) < VISIBLE_DIST_MM * KEY_RATIO && hMesh2D.
                        CheckLineForHazards(goalPt, pt2, true, normVectGoal, false,
                        NULL))
1478                {
1479                    Point* edge[2] = {&goalPt, &pt2};
1480                    WriteLineToFile("links.txt", "at", edge, 2);
1481
1482                    goalNode->AddLink(keyNodes[i]);
1483                    keyNodes[i]->AddLink(goalNode);
1484                }
1485
1486                if(!XYMatch(currentPt, pt2) && hMesh2D.CheckLineForHazards(
                        currentPt, pt2, true, normVectCurr, false, NULL))
1487                {
1488                    Point* edge[2] = {&currentPt, &pt2};
1489                    WriteLineToFile("links.txt", "at", edge, 2);
1490
1491                    currentLocNode->AddLink(keyNodes[i]);
1492                    keyNodes[i]->AddLink(currentLocNode);//As last link, means
                            less shifting when it's unlinked
1493                }
1494            }
1495
1496            if(!goalIsKeyNode && !XYMatch(currentPt, goalPt))
1497            {
1498                float dist = CalculateDirectDistance(goalPt, currentPt, true);
1499                if(dist < VISIBLE_DIST_MM * KEY_RATIO)
1500                {
1501                    Point dirVect = ZERO_POINT + (goalPt - currentPt);
1502                    Point normVect = ScaledCrossProduct(dirVect, dirVect + (Point
                            (0,0,10) - ZERO_POINT), ROBOTWIDTH/2);
1503                    if(hMesh2D.CheckLineForHazards(goalPt, currentPt, true,
                            normVect, false, NULL))
1504                    {
1505                        Point* edge[2] = {&goalPt, &currentPt};
1506                        WriteLineToFile("links.txt", "at", edge, 2);
1507
```

```
1508                        goalNode->AddLink(currentLocNode);
1509                        currentLocNode->AddLink(goalNode);
1510                    }
1511                }
1512            }
1513
1514            Point* edge[2] = {&goalPt, &(goalPt + Vector(0,2,0))};
1515            WriteLineToFile("links.txt", "at", edge, 2);
1516            edge[0] = &currentPt; edge[1] = &(currentPt - Vector(0,2,0));
1517            WriteLineToFile("links.txt", "at", edge, 2);
1518            edge[0] = &startPt; edge[1] = &(startPt - Vector(0,2,0));
1519            WriteLineToFile("links.txt", "at", edge, 2);
1520        }
1521    }
1522
1523    template<class NODE>
1524    void World<NODE>::CreateGrid()
1525    {
1526        Point normVect(0,0,0), backLeft(0,0,0), frontRight(0,0,0), currentPos
                (0,0,0);
1527        Vector hStep = Vector(MOVE_INCR, 0, 0);
1528        Vector vStep = Vector(0, MOVE_INCR, 0);
1529        currentPos = currentLocNode->GetPoint();
1530        backLeft = currentPos - Vector(VISIBLE_DIST_MM/SCALE_FACTOR,
                VISIBLE_DIST_MM/SCALE_FACTOR, 0);
1531        int numSteps = 2*(VISIBLE_DIST_MM/SCALE_FACTOR)/MOVE_INCR;
1532
1533        for(int i = 0; i < numSteps; i++)
1534        {
1535            for(int j = 0; j < numSteps; j++)
1536            {
1537                if(numHeldNodes != 0 && numHeldNodes%ARRAY_CHUNK == 0)
1538                {
1539                    void* tmp = realloc(heldNodes, (numHeldNodes + ARRAY_CHUNK) *
                            sizeof(NODE*));
1540                    if(tmp != NULL)
1541                        heldNodes = (NODE**) tmp;
1542                }
1543
1544                Face_handle tri;
1545                Point newPoint = backLeft + hStep*j + vStep*i;
1546                GetTri(&tri, &newPoint);
1547                heldNodes[numHeldNodes] = new NODE(newPoint, tri->hazardType,
                        true);
1548
1549                if(i == 0 || j == 0 || i == numSteps - 1 || j == numSteps - 1 )
1550                {
```

```
1551                    if(numEdgeNodes != 0 && numEdgeNodes%ARRAY_CHUNK == 0)
1552                    {
1553                        void* tmp = realloc(edgeNodes, (numEdgeNodes +
                                ARRAY_CHUNK) * sizeof(NODE*));
1554                        if(tmp != NULL)
1555                            edgeNodes = (NODE**) tmp;
1556                    }
1557                    edgeNodes[numEdgeNodes++] = heldNodes[numHeldNodes];
1558                }
1559                else
1560                {
1561                    if(numKeyNodes != 0 && numKeyNodes%ARRAY_CHUNK == 0)
1562                    {
1563                        void* tmp = realloc(keyNodes, (numKeyNodes + ARRAY_CHUNK)
                                * sizeof(NODE*));
1564                        if(tmp != NULL)
1565                            keyNodes = (NODE**) tmp;
1566                    }
1567                    keyNodes[numKeyNodes++] = heldNodes[numHeldNodes];
1568                }
1569
1570
1571                if(j != 0)
1572                {
1573                    heldNodes[numHeldNodes]->AddLink(heldNodes[numHeldNodes - 1])
                            ;
1574                    heldNodes[numHeldNodes - 1]->AddLink(heldNodes[numHeldNodes])
                            ;
1575                }
1576                if(i != 0)
1577                {
1578                    heldNodes[numHeldNodes]->AddLink(heldNodes[numHeldNodes -
                            numSteps]);
1579                    heldNodes[numHeldNodes - numSteps]->AddLink(heldNodes[
                            numHeldNodes]);
1580                    if(j != 0)
1581                    {
1582                        heldNodes[numHeldNodes]->AddLink(heldNodes[numHeldNodes -
                                numSteps - 1]);
1583                        heldNodes[numHeldNodes - numSteps - 1]->AddLink(heldNodes
                                [numHeldNodes]);
1584                    }
1585                    if(j != numSteps - 1)
1586                    {
1587                        heldNodes[numHeldNodes]->AddLink(heldNodes[numHeldNodes -
                                numSteps + 1]);
```

```
1588                        heldNodes[numHeldNodes - numSteps + 1]->AddLink(heldNodes
                                [numHeldNodes]);
1589                    }
1590                }
1591                numHeldNodes++;
1592            }
1593        }
1594    }
1595
1596    /** Generate MapNodes based on grid formation
1597        Have at "gridResolution" step sizes
1598        Expand based on position
1599        Update/check for changes
1600    */
1601    template<class NODE>
1602    void World<NODE>::GenGridNodes(Point dirVect)
1603    {
1604        Point normVect(0,0,0), frontLeft(0,0,0), frontRight(0,0,0), currentPos
                (0,0,0);
1605        int insertIndex = 0;
1606
1607        if(!graphInitialised)
1608        {
1609            CreateGrid();
1610            graphInitialised = true;
1611        }
1612
1613        if(dirVect.x() == 0 && dirVect.y() == 0 && dirVect.z() == 0)
1614            return;
1615
1616        dirVect =  NormaliseVector(dirVect, 10, false);
1617        normVect = ScaledCrossProduct(dirVect, dirVect + (Point(0,0,10) -
                ZERO_POINT), VISIBLE_DIST_MM);
1618        dirVect =  NormaliseVector(dirVect, VISIBLE_DIST_MM, false);
1619        currentPos = currentLocNode->GetPoint();
1620        frontLeft = currentPos + /*KEY_RATIO*/ (dirVect - normVect);
1621        frontRight = currentPos + /*KEY_RATIO*/ ((dirVect - ZERO_POINT) + (
                normVect - ZERO_POINT));
1622
1623        for(int i = 0; i < numEdgeNodes; i++)
1624        {
1625            if(edgeNodes[i] == NULL || edgeNodes[i]->GetNumLinks() == 8)
1626                continue;
1627            else if(InsideTriangleTest(edgeNodes[i]->GetPoint(), currentPos,
                    frontLeft, frontRight))
1628                ExpandNode(edgeNodes[i], i);
1629        }
```

```
1630
1631        for(int i = 0; i < numKeyNodes; i++)
1632        {
1633            Face_handle tri;
1634            if(GetTri(&tri, &keyNodes[i]->GetPoint()))
1635            {
1636                if( tri->hazardType != keyNodes[i]->hType )
1637                    keyNodes[i]->hType = tri->hazardType;
1638            }
1639        }
1640
1641        for(int i = 0; i < numEdgeNodes; i++)
1642        {
1643            if(edgeNodes[i] == NULL)
1644                continue;
1645            else if(edgeNodes[i]->GetNumLinks() == 8)
1646            {
1647                if(numKeyNodes%ARRAY_CHUNK == 0)
1648                {
1649                    void* tmp = realloc(keyNodes, (numKeyNodes + ARRAY_CHUNK) *
                            sizeof(NODE*));
1650                    if(tmp != NULL)
1651                        keyNodes = (NODE**) tmp;
1652                }
1653
1654                keyNodes[numKeyNodes++] = edgeNodes[i];
1655                edgeNodes[i] = NULL;
1656            }
1657            else
1658                edgeNodes[insertIndex++] = edgeNodes[i];
1659        }
1660
1661        if(insertIndex != numEdgeNodes)
1662        {
1663            numEdgeNodes = insertIndex;
1664            void* tmp = realloc(edgeNodes, (numEdgeNodes + ARRAY_CHUNK -
                    numEdgeNodes%ARRAY_CHUNK) * sizeof(NODE*));
1665            if(tmp != NULL)
1666                edgeNodes = (NODE**) tmp;
1667        }
1668
1669
1670        CullGridNodes(dirVect);
1671        DiscardPoints();
1672    }
1673
```

```
1674    ///TODO Cull edge nodes which are further than X distance from the current
            position?
1675    /** Update nodes within vision, keep nodes within visible dist, discard/
            ignore those further away
1676    */
1677    template<class NODE>
1678    void World<NODE>::CullGridNodes(Point dirVect)
1679    {
1680        Point normVect, frontLeft, frontRight, currentPos;
1681        //dirVect initially scaled to 10, to avoid overflow in cross product,
                then it and normVect are scaled to visible dist
1682        dirVect =  NormaliseVector(dirVect, 10, false);
1683        normVect = ScaledCrossProduct(dirVect, dirVect + (Point(0,0,10) -
                ZERO_POINT), VISIBLE_DIST_MM);
1684        dirVect =  NormaliseVector(dirVect, VISIBLE_DIST_MM, false);
1685        currentPos = currentLocNode->GetPoint();
1686        frontLeft = currentPos + KEY_RATIO * (dirVect - normVect);
1687        frontRight = currentPos + KEY_RATIO * ((dirVect - ZERO_POINT) + (normVect
                - ZERO_POINT));
1688
1689        int insertIndex = 0;
1690        for(int i = 0; i < numKeyNodes; i++)
1691        {
1692            if(keyNodes[i]->CheckForMark())
1693                continue;
1694
1695            float dist = CalculateDirectDistance(keyNodes[i]->GetPoint(),
                    currentLocNode->GetPoint(), true);
1696            if(dist < VISIBLE_DIST_MM)
1697            {
1698                if(InsideTriangleTest(keyNodes[i]->GetPoint(), currentPos,
                        frontLeft, frontRight))
1699                {
1700                    Face_handle test;
1701                    GetTri(&test, &keyNodes[i]->GetPoint());
1702                    if( test->hazardType != keyNodes[i]->hType )
1703                        keyNodes[i]->hType = test->hazardType;
1704                }
1705                keyNodes[insertIndex++] = keyNodes[i];
1706            }
1707            else
1708                keyNodes[i] = NULL;
1709            //move keyNodes[i] in to the edgeNodes or mark?
1710        }
1711
1712        if(insertIndex != numKeyNodes)
1713        {
```

```
1714            numKeyNodes = insertIndex;
1715            void* tmp = realloc(keyNodes, (numKeyNodes + ARRAY_CHUNK -
                    numKeyNodes%ARRAY_CHUNK) * sizeof(NODE*));
1716            if(tmp != NULL)
1717                keyNodes = (NODE**) tmp;
1718        }
1719
1720        insertIndex = 0;
1721        for(int i = 0; i < numEdgeNodes; i++)
1722        {
1723            if(edgeNodes[i] == NULL || edgeNodes[i]->CheckForMark())
1724                continue;
1725            else if(edgeNodes[i]->GetNumLinks() == 8)
1726                int cat = 7;
1727            else
1728                edgeNodes[insertIndex++] = edgeNodes[i];
1729        }
1730
1731        if(insertIndex != numEdgeNodes)
1732        {
1733            numEdgeNodes = insertIndex;
1734            void* tmp = realloc(edgeNodes, (numEdgeNodes + ARRAY_CHUNK -
                    numEdgeNodes%ARRAY_CHUNK) * sizeof(NODE*));
1735            if(tmp != NULL)
1736                edgeNodes = (NODE**) tmp;
1737        }
1738    }
1739
1740    template<class NODE>
1741    void World<NODE>::ExpandNode(NODE* edgeNode, int edgeInd)
1742    {
1743        float moveDist = MOVE_INCR;
1744        int numLinks = edgeNode->GetNumLinks();
1745        Vector moves[4] = {Vector(moveDist,0,0), Vector(0,0,moveDist), Vector
                (0,0,-moveDist), Vector(-moveDist,0,0)};
1746
1747        for(int i = 0; i < 4; i++)
1748        {
1749            int j = 0;
1750            for(j = 0; j < numLinks; j++)
1751            {
1752                //FIXME Bug where some linkedNodes are null. Issue of not
                        delinking on cull or ?
1753                if(edgeNode->FollowLink(j)->GetPoint() == (edgeNode->GetPoint() +
                         moves[i]))
1754                    break;
1755            }
```

```
1756            if(j == numLinks)
1757            {
1758                if(numHeldNodes%ARRAY_CHUNK == 0)
1759                {
1760                    void* tmp = realloc(heldNodes, (numHeldNodes + ARRAY_CHUNK) *
                            sizeof(NODE*));
1761                    if(tmp != NULL)
1762                        heldNodes = (NODE**) tmp;
1763                }
1764
1765                Face_handle tri;
1766                Point newPoint = edgeNode->GetPoint() + moves[i];
1767                GetTri(&tri, &newPoint);
1768                heldNodes[numHeldNodes] = new NODE(newPoint, tri->hazardType,
                        true);
1769                heldNodes[numHeldNodes]->AddLink(edgeNode);
1770                edgeNode->AddLink(heldNodes[numHeldNodes]);
1771                heldNodes[numHeldNodes]->ExpandLinks(i, moveDist, *edgeNode);
1772                //link to others
1773                //based on which of moves[] it was determines which neighbours to
                        look for
1774
1775                if(numKeyNodes != 0 && numKeyNodes%ARRAY_CHUNK == 0)
1776                {
1777                    void* tmp = realloc(keyNodes, (numKeyNodes + ARRAY_CHUNK) *
                            sizeof(NODE*));
1778                    if(tmp != NULL)
1779                        keyNodes = (NODE**) tmp;
1780                }
1781                keyNodes[numKeyNodes++] = heldNodes[numHeldNodes];
1782                numHeldNodes++;
1783            }
1784        }
1785
1786        if(edgeNode->GetNumLinks() == 8)
1787        {
1788            if(numKeyNodes != 0 && numKeyNodes%ARRAY_CHUNK == 0)
1789            {
1790                void* tmp = realloc(keyNodes, (numKeyNodes + ARRAY_CHUNK) *
                        sizeof(NODE*));
1791                if(tmp != NULL)
1792                    keyNodes = (NODE**) tmp;
1793            }
1794            keyNodes[numKeyNodes++] = edgeNodes[edgeInd];
1795            edgeNodes[edgeInd] = NULL;
1796        }
1797    }
```

```
1798
1799    /** Shows all the links associated with currently held nodes
1800        Skips marked nodes and links to nodes which are marked
1801
1802        Then overlays the links of the current linking session
1803     */
1804    template<class NODE>
1805    void World<NODE>::GetAllLinks()
1806    {
1807        WriteLineToFile("AllLinks.txt", "wt", (Point**) NULL, 0);
1808        for(int i = 0; i < numHeldNodes; i++)
1809        {
1810            if(heldNodes[i]->CheckForMark())
1811                continue;
1812
1813            for(int j = 0; j < heldNodes[i]->GetNumLinks(); j++)
1814            {
1815                if(heldNodes[i]->FollowLink(j)->CheckForMark())
1816                    continue;
1817
1818                Point* edge[2] = {&heldNodes[i]->GetPoint(), &heldNodes[i]->
                        FollowLink(j)->GetPoint()};
1819                WriteLineToFile("AllLinks.txt", "at", edge, 2);
1820            }
1821        }
1822
1823        if(SHOW_SDL) display("AllLinks.txt");
1824
1825        WriteColourChangeToFile("AllLinks.txt");
1826        FILE* latestLinks;
1827        FILE* output = fopen("AllLinks.txt", "at");
1828        latestLinks = fopen("links.txt", "rt");
1829        if (latestLinks != NULL && output != NULL)
1830        {
1831            bool colourChanged = false;
1832            char buffer[80];
1833            while(fgets(buffer, 80, latestLinks) != NULL)
1834            {
1835                if(colourChanged && buffer[0] != '\n')
1836                {
1837                    fprintf(output, "%s", buffer);
1838                }
1839                else if(buffer[0] == '-')
1840                    colourChanged = true;
1841            }
1842        }
1843        fclose(output);
```

```
1844        if(latestLinks != NULL)
1845            fclose(latestLinks);
1846    }
1847
1848    /** Shows all the links associated with current keyNodes
1849        Skips marked nodes and links to nodes which are marked
1850
1851        Then overlays the links of the current linking session
1852     */
1853    template<class NODE>
1854    void World<NODE>::GetAllKeyNodeLinks()
1855    {
1856        WriteLineToFile("AllKeyLinks.txt", "wt", (Point**) NULL, 0);
1857        for(int i = 0; i < numKeyNodes; i++)
1858        {
1859            if(heldNodes[i]->CheckForMark())
1860                continue;
1861
1862            for(int j = 0; j < keyNodes[i]->GetNumLinks(); j++)
1863            {
1864                if(keyNodes[i]->FollowLink(j)->CheckForMark())
1865                    continue;
1866
1867                Point* edge[2] = {&keyNodes[i]->GetPoint(), &keyNodes[i]->
                        FollowLink(j)->GetPoint()};
1868                WriteLineToFile("AllKeyLinks.txt", "at", edge, 2);
1869            }
1870        }
1871
1872        if(SHOW_SDL) display("AllKeyLinks.txt");
1873
1874        WriteColourChangeToFile("AllKeyLinks.txt");
1875        FILE* latestLinks;
1876        FILE* output = fopen("AllKeyLinks.txt", "at");
1877        latestLinks = fopen("links.txt", "rt");
1878        if (latestLinks != NULL && output != NULL)
1879        {
1880            bool colourChanged = false;
1881            char buffer[80];
1882            while(fgets(buffer, 80, latestLinks) != NULL)
1883            {
1884                if(colourChanged && buffer[0] != '\n')
1885                {
1886                    fprintf(output, "%s", buffer);
1887                }
1888                else if(buffer[0] == '-')
1889                    colourChanged = true;
```

```
1890            }
1891        }
1892        fclose(output);
1893        fclose(latestLinks);
1894    }
1895
1896    template<class NODE>
1897    void World<NODE>::SortNodes(NODE** mNodes, int* numMapNodes)
1898    {
1899
1900        int increment = 3;
1901        NODE* tempNode;
1902        while(increment > 0)
1903        {
1904            for(int j = 0; j < *numMapNodes; j++)
1905            {
1906                int k = j;
1907                tempNode = mNodes[j];
1908                while ((k >= increment) && (mNodes[k-increment]->GetPoint().x() <
                        tempNode->GetPoint().x()))
1909                {
1910                    mNodes[k] = mNodes[k - increment];
1911                    k = k - increment;
1912                }
1913                mNodes[k] = tempNode;
1914            }
1915            if(increment/2 != 0)
1916                increment = increment/2;
1917            else if(increment == 1)
1918                increment = 0;
1919            else
1920                increment = 1;
1921        }
1922
1923    }
1924
1925    /** Combine the presorted keyNodes and mNode arrays
1926        Should take order(n) to do so, using 2n space
1927    */
1928    template<class NODE>
1929    void World<NODE>::MergeWithKeyNodes(NODE** mNodes, int* numMapNodes, bool
            preSorted)
1930    {
1931        if(!preSorted)
1932            SortNodes(mNodes, numMapNodes);
1933
1934        NODE** tmpKeyNodes = (NODE**) calloc(numKeyNodes, sizeof(NODE*));
```

```
1935        int kInd = 0;
1936        for(int i = 0; i < numKeyNodes; i++)
1937        {
1938            if(!keyNodes[i]->CheckForMark())
1939                tmpKeyNodes[kInd++] = keyNodes[i];
1940        }
1941
1942        void* tmp = realloc(keyNodes, (kInd + *numMapNodes) * sizeof(NODE*));
1943        if(tmp != NULL)
1944            keyNodes = (NODE**) tmp;
1945
1946        int insertIndex = 0;
1947        for(int i = 0, j = 0; insertIndex < kInd + *numMapNodes; )
1948        {
1949            if(i < kInd && tmpKeyNodes[i]->CheckForMark())
1950                i++;
1951            else if(j < *numMapNodes && mNodes[j]->CheckForMark())
1952                j++;
1953            else if(i >= kInd && j >= *numMapNodes)
1954                break;
1955            else if(i < kInd && (j >= *numMapNodes || tmpKeyNodes[i]->GetPoint().
                    x() <= mNodes[j]->GetPoint().x()))
1956                keyNodes[insertIndex++] = tmpKeyNodes[i++];
1957            else
1958                keyNodes[insertIndex++] = mNodes[j++];
1959        }
1960
1961        tmp = realloc(keyNodes, (insertIndex) * sizeof(NODE*));
1962        if(tmp != NULL)
1963            keyNodes = (NODE**) tmp;
1964
1965        free(tmpKeyNodes);
1966        tmpKeyNodes = NULL;
1967        numKeyNodes = insertIndex;
1968    }
1969
1970    /** Process of testing terrain for hazardous areas.
1971        To save time only new triangles are tested, as if old ones have changed
                it will be due to a neighbouring new one. As new ones expand testing
                out to neighbours, all hazards should be identified fine.
1972
1973        Or have all non-hazard and all partial-hazard ones checked. How well do
                trust checkwall to expand out and catch all
1974    */
1975    template<class NODE> template<class TRI, class PT>
1976    void World<NODE>::IdentifyHazards(int numTri, TRI* surface)
1977    {
```

```
1978        bool allocdSurface = false;
1979        if(surface == NULL)
1980        {
1981            surface = (TRI*) calloc(ARRAY_CHUNK, sizeof(TRI));
1982            allocdSurface = true;
1983            GetAllTriangles(&surface, &numTri);
1984        }
1985
1986        for(int i = 0; i < numTri; i++)
1987        {
1988            if(surface[i]->checked)
1989                continue;
1990
1991            float dist[2] = {-1,-1};
1992            CalculateDistsToCentre(*surface[i]->GetPoint(0), *surface[i]->
                    GetPoint(1), *surface[i]->GetPoint(2), dist);
1993            if(dist[0] > HAZARD_TRI_DIST_THRESHOLD)
1994            {
1995                //can't fully trust if is a hazard as due to size could
                        erroneously detect when further data/resolution would show
                        isn't
1996                //possibly compare to neighbour triangles size?
1997                continue;
1998            }
1999
2000            if(CalculateDirectDistance( currentLocNode->GetPoint(), CalcTriCentre
                    (*surface[i]->GetPoint(0), *surface[i]->GetPoint(1), *surface[i
                    ]->GetPoint(2)), true) >= VISIBLE_DIST_MM)
2001            {
2002                continue;
2003            }
2004
2005            int angle = AngleBetweenTwoPlanes<TRI,PT>(surface[i], NULL);
2006            if(angle >= SLOPEANGLE && angle <= 180 - SLOPEANGLE)
2007            {
2008                int xDiff = FindMax(surface[i]->GetPoint(0)->x(),surface[i]->
                        GetPoint(1)->x(),surface[i]->GetPoint(1)->x()) - FindMin(
                        surface[i]->GetPoint(0)->x(),surface[i]->GetPoint(1)->x(),
                        surface[i]->GetPoint(1)->x());
2009                int yDiff = FindMax(surface[i]->GetPoint(0)->y(),surface[i]->
                        GetPoint(1)->y(),surface[i]->GetPoint(1)->y()) - FindMin(
                        surface[i]->GetPoint(0)->y(),surface[i]->GetPoint(1)->y(),
                        surface[i]->GetPoint(1)->y());
2010                float dist = sqrt(xDiff*xDiff + yDiff*yDiff);
2011                if(angle <= WALLANGLE || angle >= 180 - WALLANGLE)
2012                {
2013                    if(dist/cos(angle) >= AXLESEP)
```

```
2014                    {
2015                        surface[i]->hazardType = SLOPE;
2016                        continue;
2017                    }
2018                }
2019                else if(dist/sin(angle) >= WHEELRADIUS)
2020                {
2021                    surface[i]->hazardType = WALL;
2022                    continue;
2023                }
2024
2025                for(int j = 0; j < 3; j++)
2026                {
2027                    if(surface[i]->neighbor(j)->CheckValidTri() && !hMesh2D.
                        CheckIfInfinite(surface[i]->neighbor(j)))
2028                    {
2029                        if(surface[i]->neighbor(j)->hazardType == WALL || surface
                            [i]->neighbor(j)->hazardType == SLOPE)
2030                        {
2031                            int planesAngle = AngleBetweenTwoPlanes<TRI,PT>(
                                surface[i], surface[i]->neighbor(j));
2032                            //get cross prod of norm vectors, if z component is
                                strongest then not matching walls, the diff angle
                                 in xy view
2033                            //does it matter? maybe just if is slope neigh?
2034
2035                            if(planesAngle >= 180 - WALLANGLE)
2036                            {
2037                                if(angle < WALLANGLE || angle > 180 - WALLANGLE)
2038                                    surface[i]->hazardType = PART_WALL;
2039                                else
2040                                    surface[i]->hazardType = PART_SLOPE;
2041                            }
2042                        }
2043                        else
2044                        {
2045                            int neighAngle = AngleBetweenTwoPlanes<TRI,PT>(
                                surface[i]->neighbor(j), NULL);
2046                            if(neighAngle >= SLOPEANGLE && neighAngle <= 180 -
                                SLOPEANGLE)//this gets when neigh is either slope
                                 or another wall
2047                            {
2048                                surface[i]->checked = true;
2049                                HAZARDS hazardType = NONE;
2050                                CheckWall<TRI,PT>(&surface[i], &surface[i]->
                                    neighbor(j), &hazardType);
2051                                surface[i]->hazardType = hazardType;
```

```
2052                                }
2053                            }
2054                        }
2055                    }
2056                }
2057            }
2058
2059            for(int i = 0; i < numTri; i++)
2060            {
2061                if(surface[i]->checked)
2062                    surface[i]->checked = false;
2063                if(surface[i]->newlyCreated)
2064                    surface[i]->newlyCreated = false;
2065            }
2066
2067
2068            if(allocdSurface)
2069            {
2070                free(surface);
2071                surface = NULL;
2072            }
2073        }
2074
2075        /** Process of testing terrain for hazardous areas.
2076            To save time only new triangles are tested, as if old ones have changed
                it will be due to a neighbouring new one. As new ones expand testing
                out to neighbours, all hazards should be identified fine.
2077
2078            Or have all non-hazard and all partial-hazard ones checked. How well do
                trust checkwall to expand out and catch all
2079        */
2080        //Hazards will only be written to hazards.txt and hence visible via display
            menu if generating the nodes, preMadePoints don't write the hazards to
            file
2081        template<class NODE> template<class TRI, class PT>
2082        void World<NODE>::IdentifyKnownHazards(int numTri, TRI* surface)
2083        {
2084            bool allocdSurface = false;
2085            if(surface == NULL)
2086            {
2087                surface = (TRI*) calloc(ARRAY_CHUNK, sizeof(TRI));
2088                allocdSurface = true;
2089                GetAllTriangles(&surface, &numTri);
2090            }
2091
2092            for(int i = 0; i < numTri; i++)
2093            {
```

```
2094            float dist[2] = {-1,-1};
2095            CalculateDistsToCentre(*surface[i]->GetPoint(0), *surface[i]->
                    GetPoint(1), *surface[i]->GetPoint(2), dist);
2096            if(dist[0] > HAZARD_TRI_DIST_THRESHOLD)
2097            {
2098                continue;
2099            }
2100
2101            if(CalculateDirectDistance( currentLocNode->GetPoint(), CalcTriCentre
                    (*surface[i]->GetPoint(0), *surface[i]->GetPoint(1), *surface[i
                    ]->GetPoint(2)), true) >= VISIBLE_DIST_MM)
2102            {
2103                continue;
2104            }
2105
2106            for(int j = 0; j < numPreMadeHazards; j += 2)
2107            {
2108                if(surface[i]->hazardType != NONE)
2109                    break;
2110
2111                int x0 = preKnownHazards[j].x();
2112                int y0 = preKnownHazards[j].y();
2113                int x1 = preKnownHazards[j+1].x();
2114                int y1 = preKnownHazards[j+1].y();
2115
2116                //Or could have calcTriCentre and just check that
2117                for(int k = 0; k < 3; k++)
2118                {
2119                    int x = surface[i]->GetPoint(k)->x();
2120                    int y = surface[i]->GetPoint(k)->y();
2121                    if( ((x > x0 && x < x1) || (x < x0 && x > x1)) && ((y > y0 &&
                            y < y1) || (y < y0 && y > y1)) )
2122                    {
2123                        surface[i]->hazardType = WALL;
2124                        break;
2125                    }
2126                }
2127            }
2128        }
2129
2130        if(allocdSurface)
2131        {
2132            free(surface);
2133            surface = NULL;
2134        }
2135    }
2136
```

```
2137    /**
2138    Either called by Identify hazards to check for forming a crevasse or wall
            with other triangles
2139    */
2140    template<class NODE> template<class TRI, class PT>
2141    bool World<NODE>::CheckWall(TRI* current, TRI* testTri, HAZARDS* hazType){
2142        bool result = false;
2143
2144        float dist[2] = {-1,-1};
2145        CalculateDistsToCentre(*(*testTri)->GetPoint(0), *(*testTri)->GetPoint(1)
                , *(*testTri)->GetPoint(2), dist);
2146        if(dist[0] > HAZARD_TRI_DIST_THRESHOLD)
2147        {
2148            //possibly compare to neighbour triangles size?
2149            return result;
2150        }
2151
2152        //check depth, from top of Wall to lowest of current
2153        int xMax1 = FindMax((*current)->GetPoint(0)->x(), (*current)->GetPoint(1)
                ->x(), (*current)->GetPoint(2)->x());
2154        int xMax2 = FindMax((*testTri)->GetPoint(0)->x(), (*testTri)->GetPoint(1)
                ->x(), (*testTri)->GetPoint(2)->x());
2155        int yMax1 = FindMax((*current)->GetPoint(0)->y(), (*current)->GetPoint(1)
                ->y(), (*current)->GetPoint(2)->y());
2156        int yMax2 = FindMax((*testTri)->GetPoint(0)->y(), (*testTri)->GetPoint(1)
                ->y(), (*testTri)->GetPoint(2)->y());
2157        int zMax1 = FindMax((*current)->GetPoint(0)->z(), (*current)->GetPoint(1)
                ->z(), (*current)->GetPoint(2)->z());
2158        int zMax2 = FindMax((*testTri)->GetPoint(0)->z(), (*testTri)->GetPoint(1)
                ->z(), (*testTri)->GetPoint(2)->z());
2159        int xMin1 = FindMin((*current)->GetPoint(0)->x(), (*current)->GetPoint(1)
                ->x(), (*current)->GetPoint(2)->x());
2160        int xMin2 = FindMin((*testTri)->GetPoint(0)->x(), (*testTri)->GetPoint(1)
                ->x(), (*testTri)->GetPoint(2)->x());
2161        int yMin1 = FindMin((*current)->GetPoint(0)->y(), (*current)->GetPoint(1)
                ->y(), (*current)->GetPoint(2)->y());
2162        int yMin2 = FindMin((*testTri)->GetPoint(0)->y(), (*testTri)->GetPoint(1)
                ->y(), (*testTri)->GetPoint(2)->y());
2163        int zMin1 = FindMin((*current)->GetPoint(0)->z(), (*current)->GetPoint(1)
                ->z(), (*current)->GetPoint(2)->z());
2164        int zMin2 = FindMin((*testTri)->GetPoint(0)->z(), (*testTri)->GetPoint(1)
                ->z(), (*testTri)->GetPoint(2)->z());
2165
2166        Coord max, min;
2167        max.xVal = FindMax(xMax1, xMax2, xMin1);
2168        max.yVal = FindMax(yMax1, yMax2, yMin1);
2169        max.zVal = FindMax(zMax1, zMax2, zMin1);
```

```
2170            min.xVal = FindMin(xMin1, xMin2, xMax1);
2171            min.yVal = FindMin(yMin1, yMin2, yMax1);
2172            min.zVal = FindMin(zMin1, zMin2, zMax1);
2173
2174    ///TODO Improve how the overall angle/steepness of two triangles is calculated
2175
2176            int combinedDist = CalculateDirectDistance(max, min, false);
2177            Coord steepestVector = max - min;
2178            Coord levelVector = steepestVector;
2179            levelVector.zVal = 0;
2180            int angle = AngleBetweenTwoVectors(steepestVector, levelVector);
2181
2182            if(angle >= WALLANGLE && angle <= 180 - WALLANGLE && combinedDist/sin(
                    angle) >= WHEELRADIUS)
2183            {
2184                (*current)->hazardType = WALL;
2185                *hazType = WALL;
2186                if((*testTri)->hazardType == NONE || (*testTri)->hazardType ==
                        PART_WALL || (*testTri)->hazardType == PART_SLOPE)
2187                    (*testTri)->hazardType = WALL;
2188                result = true;
2189            }
2190            else if(angle >= SLOPEANGLE && angle <= 180 - SLOPEANGLE && combinedDist/
                    cos(angle) >= AXLESEP)
2191            {
2192                (*current)->hazardType = SLOPE;
2193                *hazType = SLOPE;
2194                if((*testTri)->hazardType == NONE || (*testTri)->hazardType ==
                        PART_WALL || (*testTri)->hazardType == PART_SLOPE)
2195                    (*testTri)->hazardType = SLOPE;
2196                result = true;
2197            }
2198            else if(combinedDist/cos(angle) < AXLESEP)
2199            {
2200                for(int i = 0; i < 3 && result == false; i++)
2201                {
2202                    if((*testTri)->neighbor(i)->CheckValidTri() && !hMesh2D.
                            CheckIfInfinite((*testTri)->neighbor(i)) && !(*testTri)->
                            neighbor(i)->checked)
2203                    {
2204                        bool hazards = false;
2205                        if((*testTri)->neighbor(i)->hazardType == WALL || (*testTri)
                                ->neighbor(i)->hazardType == SLOPE)
2206                        {
2207                            if(angle >= WALLANGLE && angle <= 180 - WALLANGLE)
2208                                *hazType = WALL;
2209                            else if(angle >= SLOPEANGLE && angle <= 180 - SLOPEANGLE)
```

```
2210                            *hazType = SLOPE;
2211                        }
2212                        else
2213                        {
2214                            int neighAngle = AngleBetweenTwoPlanes<TRI,PT>((*testTri)
                                ->neighbor(i), NULL);
2215                            if(neighAngle >= SLOPEANGLE && neighAngle <= 180 -
                                SLOPEANGLE)//this gets when neigh is either slope or
                                another wall
2216                            {
2217                                (*testTri)->neighbor(i)->checked = true;
2218                                hazards = CheckWall<TRI,PT>(current, &(*testTri)->
                                    neighbor(i), hazType);
2219                            }
2220                        }
2221
2222                        if(*hazType != NONE)
2223                        {
2224                            (*testTri)->hazardType = *hazType;
2225                            result = true;
2226                            break;
2227                        }
2228                    }
2229                }
2230            }
2231
2232            return result;
2233        }
2234
2235        /** True if a hazard was encountered
2236         */
2237        template<class NODE>
2238        bool World<NODE>::CheckBetweenPoints(Point start, Point end, bool
                checkParallel, Point normVect)
2239        {
2240            return hMesh2D.CheckLineForHazards(start, end, checkParallel, normVect,
                    false, NULL);
2241        }
2242
2243        template<class NODE>
2244        bool World<NODE>::CheckBetweenHazardNodes(Point pt1, Point pt2, bool* isSlope
                )
2245        {
2246            // Checks straight line is not blocked
2247            bool result = false;
2248            Point dirVect = ZERO_POINT + (pt2 - pt1);
2249            Point start = pt1 + (NormaliseVector(dirVect, 2.0f, false) - ZERO_POINT);
```

```
2250            Point end = pt2 - (NormaliseVector(dirVect, 2.0f, false) - ZERO_POINT);
2251
2252            Face_handle encompTri = Face_handle();
2253            if( ! ((GetTri(&encompTri, &start) && encompTri->hazardType) || (GetTri(&
                    encompTri, &end) && encompTri->hazardType)) )
2254            {
2255                //if(CheckClearPath<Face_handle, Point>(start, end, dirVect,
                        ZERO_POINT, false, 0))
2256                Point dirVect = ZERO_POINT + (pt2 - pt1);
2257                Point normVect = CrossProduct(dirVect, dirVect + (Point(0,0,10) -
                        ZERO_POINT));
2258                normVect = NormaliseVector<Point>(normVect , ROBOTWIDTH/2, false);
2259                if(hMesh2D.CheckLineForHazards(start, end, false, normVect, true,
                        isSlope))
2260                    result = true;
2261            }
2262            return result;
2263        }
2264
2265
2266    #endif /*WORLD_H_*/
```

A.8.2 World.cpp

```
1   #include "adv_math.h"
2   #include "World.h"
3
4       Hazard::Hazard()
5       {
6           numPoints = 0;
7           points = (Point*) calloc(LIST_CHUNK, sizeof(Point));
8       }
9
10      void Hazard::AddPoint(Point newPt)
11      {
12          int i;
13          for(i = numPoints - 1; i >= 0; i--)//as more likely that recently added
                points match
14          {
15              if(points[i] == newPt)
16                  break;
17          }
18
19          if(i < 0)
20          {
21              if(numPoints != 0 && numPoints%LIST_CHUNK == 0)
```

```
22              {
23                  void* tmp = realloc(points, (numPoints + LIST_CHUNK) * sizeof(
                        Point));
24                  if(tmp != NULL)
25                      points = (Point*) tmp;
26                  else
27                      if(SHOW_ERRORS) printf("Error - Could not reallocate memory
                            for new point in hazard/n");
28              }
29
30              points[numPoints++] = newPt;
31          }
32      }
33
34      void Hazard::AdjoinHazard(Hazard* otherHazard)
35      {
36          Point* otherPoints = otherHazard->GetPoints();
37          for(int i = 0; i < otherHazard->GetNumPoints(); i++)
38          {
39              AddPoint(otherPoints[i]);
40          }
41      }
42
43      bool Hazard::CompareHazard(Hazard* otherHazard)
44      {
45          Point* otherPoints = otherHazard->GetPoints();
46          for(int i = 0; i < numPoints; i++)
47          {
48              for(int j = 0; j < otherHazard->GetNumPoints(); j++)
49              {
50                  if(points[i] == otherPoints[j])//Is a single matching point
                        enough to base a join on?
51                      return true;
52              }
53          }
54
55          return false;
56      }
```

A.9 adv_math

A.9.1 adv_math.h

```
1  /*************************************************************************
2   *   Copyright (C) 2007 by Michael Douglas Hasler   *
```

```
3  *    Redheadpunk@gmail.com    *
4  *                                                                    *
5  *    This program is free software; you can redistribute it and/or modify  *
6  *    it under the terms of the GNU General Public License as published by  *
7  *    the Free Software Foundation; either version 2 of the License, or     *
8  *    (at your option) any later version.                                   *
9  *                                                                    *
10 *    This program is distributed in the hope that it will be useful,       *
11 *    but WITHOUT ANY WARRANTY; without even the implied warranty of        *
12 *    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the         *
13 *    GNU General Public License for more details.                          *
14 *                                                                    *
15 *    You should have received a copy of the GNU General Public License     *
16 *    along with this program; if not, write to the                         *
17 *    Free Software Foundation, Inc.,                                       *
18 *    59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.             *
19 **********************************************************************/
20 #ifndef ADV_MATH_H_
21 #define ADV_MATH_H_
22
23 #include <stdio.h>
24 #include <stdlib.h>
25 #include <math.h>
26 #include "Display.h"
27 #include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
28 typedef CGAL::Exact_predicates_inexact_constructions_kernel::Vector_3    Vector;
29 typedef CGAL::Exact_predicates_inexact_constructions_kernel::Point_3    Point;
30
31 const bool  SHOW_MATH_ERRORS = false;
32 const bool  SHOW_MATH_WARNINGS = false;
33 const bool  SHOW_ITEM_ERRORS = false;
34 const bool  SHOW_ITEM_WARNINGS = false;
35
36 #define PI_31DP 3.14159265358979323846264338327 95
37 #define ARRAY_CHUNK 60
38
39 #ifndef E_HAZARD
40 #define E_HAZARD
41
42 enum HAZARDS{
43     MIN_HAZARD = -1,
44     NONE = 0,
45     SLOPE,
46     WALL,
47     CREVASSE,
48     //To identify small chunks which have wall or slope angles but may form the
           opposite when neighbours considered
```

```
49      PART_WALL,
50      PART_SLOPE,
51      MAX_HAZARD
52      };
53  #endif
54
55  enum {  OUTSIDE = 0,
56      INSIDE,
57      ON_VERTEX,
58      ON_EDGE,
59      ABOVE_EDGE
60      };
61
62  struct Coord
63  {
64      Coord(){ xVal = 0; yVal = 0; zVal = 0;}
65      Coord(int setX, int setY, int setZ);
66      bool operator==(const Coord &other);
67      bool operator!=(const Coord &other);
68      Coord operator+= (const Coord &other);
69      Coord operator+ (const Coord &other);
70      Coord operator- (const Coord &other);
71      bool CheckVert(Coord point2, Coord point3);
72      Coord CrossProduct(Coord otherCoord);
73      long DotProduct(Coord otherCoord);
74      bool CoordsVertOffset(Coord otherCoord);
75      float CalculateDirectDistance(Coord otherCoord, bool XYMode);
76      int AngleBetweenTwoVectors(Coord otherCoord);
77
78      ///For compatibility/interchangeability with Point_3
79      int x() { return xVal; }
80      int y() { return yVal; }
81      int z() { return zVal; }
82
83      int xVal, yVal, zVal;
84  };
85      Coord FillCoord(int xVal, int yVal, int zVal, int scale);
86
87  struct Triangle
88  {
89      Triangle();
90      Triangle(Coord* pointA, Coord* pointB, Coord* pointC);
91      void DelinkTriangle();
92      bool CheckValidTri();
93      bool CheckVert();
94      bool CheckIfNeighbours(Triangle* test);
95      bool TestNeighboursCorrect();
```

```
96      void SetNeighbour(Triangle* neigh, int num);
97
98      ///For compatibility/interchangeability with faces
99      Triangle* neighbor(int num);
100     Coord* GetPoint(int num);
101
102     void CommonPoints(Triangle* otherTri, int* ptsA, int* ptsB, int* notCommonA,
            int* notCommonB);
103     void LinkTwoTriangles(Triangle* otherTri);
104     bool SameTri(Triangle* triToTest, bool identicalMatchesOnly, bool
            modDiffInstanceSameTri);
105     bool SameTriNeighboursCheck(Triangle* triToTest);
106     bool SameTriSameNeighbourCheck(Triangle* triToTest, Triangle*
            triNeighbourToTest);
107
108     void RecursGetTri(Triangle*** allTriangles, int *numTri);
109
110 //  int DistanceAcrossATriangle(Triangle* neighbour);
111     int AngleBetweenTwoPlanes(Triangle* neighbourTris);
112     int InterpolateZValue(Coord retrievalPoint);
113
114     Coord* point1;
115     Coord* point2;
116     Coord* point3;
117     Triangle* neighbour1;
118     Triangle* neighbour2;
119     Triangle* neighbour3;
120     HAZARDS hazardType;
121     bool checked;
122     bool vert;
123 };
124
125 struct Edge
126 {
127     Edge(){ pointA = NULL; pointB = NULL; leftSideDone = false; rightSideDone =
            false; }
128     Edge(Coord* pointA, Coord* pointB, bool left, bool right, Triangle* tri);
129     bool SameEdge(Edge* newEdge, bool modEdges);
130     Coord *pointA, *pointB;
131     bool leftSideDone, rightSideDone;
132     Triangle* sideTri;
133 };
134
135 static const void* RESERVED = (void*) (NULL + 0x500);
136 //static const void* RESERVED_UPPER = (void*) (NULL + 0x100000000000);
137
138 ///Math etc
```

```
139 template <class PT>
140 PT FindMax(const PT& a, const PT& b, const PT& c){
141     if(a > b && a > c)
142         return a;
143     else if (b > c)
144         return b;
145     else
146         return c;
147 }
148
149 template <class PT>
150 PT FindMin(const PT& a, const PT& b, const PT& c){
151     if(a < b && a < c)
152         return a;
153     else if (b < c)
154         return b;
155     else
156         return c;
157 }
158
159 template <class PT>
160 PT AddPoints(PT lhs, PT rhs)
161 {
162     PT C(lhs.x() + rhs.x(), lhs.y() + rhs.y(), lhs.z() + rhs.z());
163     return C;
164 }
165
166 template <class PT>
167 PT SubPoints(PT lhs, PT rhs)
168 {
169     PT C(lhs.x() - rhs.x(), lhs.y() - rhs.y(), lhs.z() - rhs.z());
170     return C;
171 }
172
173 template <class PT>
174 PT NormaliseVector(PT vector, float level, bool allowLess)
175 {
176     PT nVector;
177     float magn = CalculateDirectDistance(PT(0,0,0), vector, true);
178     if(magn != 0)
179     {
180         if(magn >= level || !allowLess)//instances where don't want to be less
                than level
181             nVector = PT(vector.x() * level / magn, vector.y() * level / magn,
                    vector.z() * level / magn);
182         else
183             nVector = PT(vector.x(), vector.y(), vector.z());
```

```
184     }
185     else
186     {
187         if(SHOW_MATH_ERRORS) printf("Error - Magnitude of vector was zero\n");
188         nVector = PT(0,0,0);
189     }
190
191     return nVector;
192 }
193
194 template <class PT>
195 PT CrossProduct(PT coord, PT otherCoord)
196 {
197     PT C(coord.y() * otherCoord.z() - coord.z() * otherCoord.y(),
198         -coord.x() * otherCoord.z() + coord.z() * otherCoord.x(),
199         coord.x() * otherCoord.y() - coord.y() * otherCoord.x());
200     return C;
201 }
202
203 template <class PT>
204 PT ScaledCrossProduct(PT coord, PT otherCoord, int magnitude)
205 {
206     PT C(coord.y() * otherCoord.z() - coord.z() * otherCoord.y(),
207         -coord.x() * otherCoord.z() + coord.z() * otherCoord.x(),
208         coord.x() * otherCoord.y() - coord.y() * otherCoord.x());
209     C = NormaliseVector(C, magnitude, false);
210     return C;
211 }
212
213 template <class PT>
214 long DotProduct(PT coord, PT otherCoord)
215 {
216     long C = coord.x() * otherCoord.x() + coord.y() * otherCoord.y() + coord.z()
            * otherCoord.z();
217     return C;
218 }
219
220 template <class PT>
221 PT operator+= (PT lhs, const PT &other)
222 {
223     PT C = PT(lhs.x() + other.x(), lhs.y() + other.y(), lhs.z() + other.z());
224     return C;
225 }
226
227 template <class TYPE1, class TYPE2>
228 TYPE2 ConvertPointType(TYPE1 point)
229 {
```

```
230     TYPE2 converted(point.x(), point.y(), point.z());
231     return converted;
232 }
233
234 template <class PT>
235 bool ToTheLeftOfVector(PT lineAB, PT currentPos, PT testPoint){
236     bool result = false;
237     if((lineAB.x()) * (testPoint.y() - currentPos.y()) - (testPoint.x() -
            currentPos.x()) * (lineAB.y()) > 0 )
238         result = true;
239     return result;
240 }
241
242 template <class PT>
243 bool ToTheLeft(PT linePtA, PT linePtB, PT testVector){
244     bool result = false;
245     if((linePtB.x() - linePtA.x()) * (testVector.y() - linePtA.y()) - (testVector
            .x() - linePtA.x()) * (linePtB.y() - linePtA.y()) > 0 )
246         result = true;
247     return result;
248 }
249
250 template <class PT>
251 bool SameSide(PT linePtA, PT linePtB, PT comparePoint, PT testPoint){
252     bool result = false;
253     if(((linePtB.x() - linePtA.x()) * (testPoint.y() - linePtA.y()) - (testPoint.
            x() - linePtA.x()) * (linePtB.y() - linePtA.y())) *
254     ((linePtB.x() - linePtA.x()) * (comparePoint.y() - linePtA.y()) - (
            comparePoint.x() - linePtA.x()) * (linePtB.y() - linePtA.y())) > 0)
255         result = true;
256     return result;
257 }
258
259 template <class PT>
260 bool XYMatch(PT lhs, PT other)
261 {
262     if(lhs.x() == other.x() && lhs.y() == other.y())
263         return true;
264     else
265         return false;
266 }
267
268 template <class TRI, class PT>
269 int AngleBetweenTwoPlanes(TRI tri, TRI neighbourTris)
270 {
271     int result = 0;
272     PT normA(0,0,0), normB(0,0,0);
```

```
273     PT v0(0,1,0);
274     PT v1(1,0,0);
275     PT v2(0,1,0);
276     PT v3(1,0,0);
277
278     if(!tri->CheckValidTri())
279     {
280         if(SHOW_MATH_ERRORS) printf("Error - Tri is not valid\n");
281         return -1;
282     }
283     else
284     {
285         v0 = SubPoints<PT>(*(tri->GetPoint(1)), *(tri->GetPoint(0)));
286         v1 = SubPoints<PT>(*(tri->GetPoint(2)), *(tri->GetPoint(0)));
287
288         if(neighbourTris != NULL)
289         {
290             v2 = SubPoints<PT>(*neighbourTris->GetPoint(1), *neighbourTris->
                    GetPoint(0));
291             v3 = SubPoints<PT>(*neighbourTris->GetPoint(2), *neighbourTris->
                    GetPoint(0));
292         }
293
294         normA = CrossProduct<PT>(v0, v1);
295         normB = CrossProduct<PT>(v2, v3);
296
297         int numer = DotProduct<PT>(normA, normB);
298         double a = normA.x()*normA.x() + normA.y()*normA.y() + normA.z()*normA.z
                ();
299         double b = normB.x()*normB.x() + normB.y()*normB.y() + normB.z()*normB.z
                ();
300         long denom;
301
302         if(a < 0 || b < 0)
303         {
304             double sqrtA, sqrtB;
305             if(SHOW_MATH_WARNINGS) printf("Warning - Numerical overflow, negative
                    number encountered. Countering of this may result in some
                    inaccuracy\n");
306             if(a < 0)
307             {
308                 a = (normA.x()/100)*(normA.x()/100) + (normA.y()/100)*(normA.y()
                        /100) + (normA.z()/100)*(normA.z()/100);
309                 sqrtA = sqrt(a)*100;
310             }
311             else
312                 sqrtA = sqrt(a);
```

```
313             if(b < 0)
314             {
315                 a = (normB.x()/100)*(normB.x()/100) + (normB.y()/100)*(normB.y()
                        /100) + (normB.z()/100)*(normB.z()/100);
316                 sqrtB = sqrt(b)*100;
317             }
318             else
319                 sqrtB = sqrt(b);
320             denom = sqrtA * sqrtB;
321         }
322         else
323             denom = sqrt(a) * sqrt(b);
324         if(denom > 0)
325             result =  acos((float) numer / (float) denom) * (180/PI_31DP);
326         else
327             if(SHOW_MATH_ERRORS) printf("Error - denominator is zero\n");
328     }
329
330     return result;
331 }
332
333 template <class PT>
334 int AngleBetweenTwoVectors(PT v0, PT v1)
335 {
336     int dotProd = DotProduct(v0, v1);
337     float denom = sqrt(v0.x()*v0.x() + v0.y()*v0.y() + v0.z()*v0.z()) * sqrt(v1.x
            ()*v1.x() + v1.y()*v1.y() + v1.z()*v1.z());
338     if(denom > 0)
339     {
340         return acos(dotProd/denom) * (180/PI_31DP);
341     }
342     else
343     {
344         if(SHOW_MATH_ERRORS) printf("Error - denominator is zero\n");
345         return 0;
346     }
347 }
348
349 /** Calculates distance
350     * Given two points, it returns a direct line distance between these
351     * If the second point given is null, then the straightline distance to the
            goal is calculated
352     */
353 template <class PT>
354 float CalculateDirectDistance(PT startCoord, PT otherCoord, bool XYMode)
355 {
356     float distance = 0;
```

```
357     double sqrNum;
358     double xDiff = otherCoord.x() - startCoord.x();
359     double yDiff = otherCoord.y()-startCoord.y();
360     double zDiff = otherCoord.z() - startCoord.z();
361
362     if(XYMode)
363         sqrNum = xDiff*xDiff + yDiff*yDiff;
364     else
365         sqrNum = xDiff*xDiff + yDiff*yDiff + zDiff*zDiff;
366     distance = sqrt(sqrNum);
367
368     if(distance <= 0  && startCoord == otherCoord)
369     {
370         return 0;    //or less than one, as that indicates error
371     }
372     else
373         return distance;
374 }
375
376 template <class TRI, class PT>
377 int DistanceAcrossATriangle(TRI tri, TRI neighbour)
378 {
379     PT v0,v1;
380     PT *adjoiningEdge[2], pointB;
381     int a = 0;
382     int theta, dist, hypot;
383
384     if(tri->GetPoint(0) != neighbour->GetPoint(0) && tri->GetPoint(0) !=
          neighbour->GetPoint(1) && tri->GetPoint(0) != neighbour->GetPoint(2))
385     {
386         pointB = *tri->GetPoint(0);
387         adjoiningEdge[a++] = tri->GetPoint(1);
388         adjoiningEdge[a++] = tri->GetPoint(2);
389     }
390     else if(tri->GetPoint(1) != neighbour->GetPoint(0) && tri->GetPoint(1) !=
          neighbour->GetPoint(1) && tri->GetPoint(1) != neighbour->GetPoint(2))
391     {
392         adjoiningEdge[a++] = tri->GetPoint(0);
393         pointB = *tri->GetPoint(1);
394         adjoiningEdge[a++] = tri->GetPoint(2);
395     }
396     else
397     {
398         adjoiningEdge[a++] = tri->GetPoint(0);
399         adjoiningEdge[a++] = tri->GetPoint(1);
400         pointB = *tri->GetPoint(2);
401     }
```

```
402
403     v0 = SubPoints(*adjoiningEdge[0],*adjoiningEdge[1]);
404     v1 = SubPoints(*adjoiningEdge[0], pointB);
405
406     hypot = CalculateDirectDistance<PT>(*adjoiningEdge[0], pointB, false);
407     theta = AngleBetweenTwoVectors<PT>(v0, v1);
408     dist = hypot*sin(theta * (PI_31DP/180));
409
410     return dist;
411 }
412
413 template <class PT>
414 int InsideTriangleTest(PT point, PT triA, PT triB, PT triC)
415 {
416     int result;
417 /* If the point is outside the triangle's bounding box, can skip any further
        testing, else do barycentric test */
418     if ((point.x() > FindMax<int>(triA.x(), triB.x(), triC.x())) || (point.y() >
          FindMax<int>(triA.y(), triB.y(), triC.y())) ||
419         (point.x() < FindMin<int>(triA.x(), triB.x(), triC.x())) || (point.y() <
              FindMin<int>(triA.y(), triB.y(), triC.y())))
420     {
421         result = OUTSIDE;
422     }
423     else
424     {
425         //Possibly have these scaled down by 10?
426         unsigned long xPt  = point.x(), x1 = triA.x(), x2 = triB.x(), x3 = triC.x
              (), yPt = point.y(), y1 = triA.y(), y2 = triB.y(), y3 = triC.y();
427
428         //calculate area of a triangle
429         unsigned long area = abs((x2*y1 - x1*y2)+(x3*y2 - x2*y3)+(x1*y3 - x3*y1))
              ;
430         long area1 = abs((x2*yPt - xPt*y2)+(x3*y2 - x2*y3)+(xPt*y3 - x3*yPt));
431         long area2 = abs((xPt*y1 - x1*yPt)+(x3*yPt - xPt*y3)+(x1*y3 - x3*y1));
432         long area3 = abs((x2*y1 - x1*y2)+(xPt*y2 - x2*yPt)+(x1*yPt - xPt*y1));
433
434         if(area1 < 0 || area2 < 0 || area3 < 0)
435         //while(area1 < 0 || area2 < 0 || area3 < 0)
436         {
437             //could have some test to determine how much to scale by - based on
                  magnitude of smallest number, or just get it looping
438             xPt = xPt/10, x1 = x1/10, x2 = x2/10, x3 = x3/10, yPt = yPt/10, y1 =
                  y1/10, y2 = y2/10, y3 = y3/10;
439             area = abs((x2*y1 - x1*y2)+(x3*y2 - x2*y3)+(x1*y3 - x3*y1));
440             area1 = abs((x2*yPt - xPt*y2)+(x3*y2 - x2*y3)+(xPt*y3 - x3*yPt));
441             area2 = abs((xPt*y1 - x1*yPt)+(x3*yPt - xPt*y3)+(x1*y3 - x3*y1));
```

```
442             area3 = abs((x2*y1 - x1*y2)+(xPt*y2 - x2*yPt)+(x1*yPt - xPt*y1));
443         }
444
445         if(area1 + area2 + area3 > area)
446             result = OUTSIDE;
447         else
448         {
449             //Set z's to zero so vertically translated points also caught, since
                    their XY area is same
450             PT tPoint, tTriA, tTriB, tTriC;
451             tPoint = PT(point.x(), point.y(), 0); tTriA = PT(triA.x(), triA.y(),
                    0); tTriB = PT(triB.x(), triB.y(), 0); tTriC = PT(triC.x(), triC.
                    y(), 0);
452             //if(point, triA) || point, triB) || point, triC))
453             if(tPoint == tTriA || tPoint == tTriB || tPoint == tTriC)
454                 result = ON_VERTEX;
455             else if(CheckAlongEdge(true, point, triA, triB) || CheckAlongEdge(
                    true, point, triA, triC) || CheckAlongEdge(true, point, triB,
                    triC))
456                 result = ON_EDGE;
457             else if(CheckAlongEdge(false, point, triA, triB) || CheckAlongEdge(
                    false, point, triA, triC) || CheckAlongEdge(false, point, triB,
                    triC))
458                 result = ABOVE_EDGE;
459             else
460                 result = INSIDE;
461         }
462     }
463
464     return result;
465 }
466
467 /** Centroids, circumcenters and inncer circles were considered as the means of
        determining a central point, the centroid was chosen
468 */
469 template <class PT>
470 PT CalcTriCentre(PT triA, PT triB, PT triC)
471 {
472     PT v0, v1, p0, p1;
473     v0 = PT(0,0,0) + (triC - triB);
474     p0 = PT(triB.x() + v0.x()/2, triB.y() + v0.y()/2, triB.z() + v0.z()/2);
475     v1 = PT(0,0,0) + (p0 - triA);
476     p1 = PT(triA.x() + v1.x()*2/3, triA.y() + v1.y()*2/3, triA.z() + v1.z()*2/3);
477     return p1;
478 }
479
480 template <class PT>
```

```
481 void CalculateDistsToCentre(PT coord1, PT coord2, PT coord3, float* dists)
482 {
483     //For skinny triangles, the centre x,y,z may seem to be same as a coord due
            to rounding error
484     PT v0;
485     v0 = coord3 - (coord2 - PT(0,0,0));
486     float x = v0.x();
487     float y = v0.y();
488     float z = v0.z();
489     x = coord1.x() + (coord2.x() + x/2 - coord1.x())*2/3;
490     y = coord1.y() + (coord2.y() + y/2 - coord1.y())*2/3;
491     z = coord1.z() + (coord2.z() + z/2 - coord1.z())*2/3;
492     //CalcTriCentre(coord1, coord2, coord3);
493
494     float distance1, distance2, distance3;
495
496     double sqrNum1 = (x - coord1.x())*(x - coord1.x()) + (y-coord1.y())*(y-coord1
            .y()) + (z - coord1.z())*(z - coord1.z());
497     distance1 = sqrt(sqrNum1);
498     double sqrNum2 = (x - coord2.x())*(x - coord2.x()) + (y-coord2.y())*(y-coord2
            .y()) + (z - coord2.z())*(z - coord2.z());
499     distance2 = sqrt(sqrNum2);
500     double sqrNum3 = (x - coord3.x())*(x - coord3.x()) + (y-coord3.y())*(y-coord3
            .y()) + (z - coord3.z())*(z - coord3.z());
501     distance3 = sqrt(sqrNum3);
502
503     if(distance1 <= 0  || distance2 <= 0 || distance3 <= 0)
504     {
505         if(SHOW_MATH_ERRORS) printf("Error - Invalid distance was calculated\n");
506     }
507
508     dists[0] = FindMax<float>(distance1,distance2,distance3);
509     dists[1] = FindMin<float>(distance1,distance2,distance3);
510 }
511
512 /** Checks the testPoint lies on the line from pt1 to pt2, but is not either pt1
        or pt2 */
513 template <class PT>
514 bool CheckPointInLine(PT point1, PT point2, double x, double y)
515 {
516     bool result = false;
517     if( ! ((point1.x() == x && point1.y() == y) || (point2.x() == x && point2.y()
             == y)) )
518     {
519         int x1_max = FindMax<int>(point1.x(), point2.x(), 0);
520         int x1_min = FindMin<int>(point1.x(), point2.x(), x1_max);
521         int y1_max = FindMax<int>(point1.y(), point2.y(), 0);
```

```
522         int y1_min = FindMin<int>(point1.y(), point2.y(), y1_max);
523         if(x <= x1_max && x >= x1_min && y <= y1_max && y >= y1_min)
524             result = true;
525     }
526
527     return result;
528 }
529
530
531 /** Lower is used so, can return which is lower when is a part of a if/logic
        test
532 * If orthSharedVert is true then point1 should be the point shared by the
        triangles which the two edges come from
533 */
534 template <class PT>
535 bool CheckEdgeIntersections(bool samePlane, bool orthSharedVert, PT point1, PT
        point2, PT point3, PT point4, int* lower)
536 {
537     bool result = false;
538     double A1 = point2.y() - point1.y();
539     double B1 = point1.x() - point2.x();
540     double C1 = A1*point1.x() + B1*point1.y();
541
542     double A2 = point4.y() - point3.y();
543     double B2 = point3.x() - point4.x();
544     double C2 = A2*point3.x() + B2*point3.y();
545
546     double det = A1*B2 - A2*B1;
547     if(det != 0)
548     {
549         double x = (B2*C1 - B1*C2)/det;
550         double y = (A1*C2 - A2*C1)/det;
551         if(orthSharedVert && ((point1.x() == x && point1.y() == y) || (point2.x()
                == x && point2.y() == y) || (point3.x() == x && point3.y() == y) ||
              (point4.x() == x && point4.y() == y)) )
552             result = false;//intersection is a shared point
553         else if(CheckPointInLine(point1, point2, x,y) && CheckPointInLine(point3,
                point4, x,y)) // Issue of precision loss?
554         {
555             result = true;
556             if(lower != NULL)
557             {
558                 if(samePlane)
559                     *lower = 1;///perhaps having this be CalcBetterTri is a
                          better option
560                 else
561                 {
```

```
562                     int x1_max = FindMax<int>(point1.x(), point2.x(), 0);
563                     int x1_min = FindMin<int>(point1.x(), point2.x(), x1_max);
564                     int x2_max = FindMax<int>(point3.x(), point4.x(), 0);
565                     int x2_min = FindMin<int>(point3.x(), point4.x(), x2_max);
566                     float z1_max = FindMax<int>(point1.z(), point2.z(), 0);
567                     float z1_min = FindMin<int>(point1.z(), point2.z(), z1_max);
568                     float z2_max = FindMax<int>(point3.z(), point4.z(), 0);
569                     float z2_min = FindMin<int>(point3.z(), point4.z(), z2_max);
570                     double z1 = z1_min + (z1_max - z1_min) * (x / (x1_max -
                            x1_min));
571                     double z2 = z2_min + (z2_max - z2_min) * (x / (x2_max -
                            x2_min));
572                     if(z1 < z2)
573                         *lower = 1;
574                     else if(z1 > z2)
575                         *lower = 2;
576                 }
577             }
578         }
579     }
580
581     return result;
582 }
583
584 /** Gets intersecting point vectors known to converge and checks it isn't a
        shared vertex */
585 template <class PT>
586 bool GetIntersection(PT point1, PT point2, PT point3, PT point4, PT* inter)
587 {
588     bool result = false;
589     double A1 = point2.y() - point1.y();
590     double B1 = point1.x() - point2.x();
591     double C1 = A1*point1.x() + B1*point1.y();
592
593     double A2 = point4.y() - point3.y();
594     double B2 = point3.x() - point4.x();
595     double C2 = A2*point3.x() + B2*point3.y();
596
597     double det = A1*B2 - A2*B1;
598     double x = (B2*C1 - B1*C2)/det;
599     double y = (A1*C2 - A2*C1)/det;
600
601     if( ! ((point1.x() == x && point1.y() == y) || (point2.x() == x && point2.y()
            == y) || (point3.x() == x && point3.y() == y) || (point4.x() == x &&
           point4.y() == y)) )
602     {
603         *inter = PT(x,y,0);
```

```
604         result = true;
605     }
606
607     return result;
608 }
609
610 int CalcBetterTriangle(Coord tri1_A, Coord tri1_B, Coord tri1_C, Coord tri2_A,
        Coord tri2_B, Coord tri2_C);
611 // float* CalculateDistsToCentre(Coord coord1, Coord coord2, Coord coord3);
612 int CheckTriangleEdgeIntersections(bool samePlane, Coord tri1_A, Coord tri1_B,
        Coord tri1_C, Coord tri2_A, Coord tri2_B, Coord tri2_C);
613 Triangle** LinkTriangles(Triangle** newTris, int triIndex, int numEdge);
614 Triangle** SortTriangles(Triangle** newTris, int triIndex);
615
616 ///EdgeHugger
617 template <class PT>
618 bool CheckAlongEdge(bool checkZ, PT testPoint, PT point1, PT point2)
619 {
620     bool result = false;
621     int x1_max = FindMax<int>(point1.x(), point2.x(), 0);
622     int x1_min = FindMin<int>(point1.x(), point2.x(), x1_max);
623     int y1_max = FindMax<int>(point1.y(), point2.y(), 0);
624     int y1_min = FindMin<int>(point1.y(), point2.y(), y1_max);
625     if(testPoint.x() <= x1_max && testPoint.x() >= x1_min && testPoint.y() <=
        y1_max && testPoint.y() >= y1_min)
626     {
627         if(!checkZ)
628             testPoint += PT(0,0, -testPoint.z()); point1 += PT(0,0, -point1.z());
                  point2 += PT(0,0, -point2.z());
629         PT cross = CrossProduct((testPoint - (point1 - PT(0,0,0))), (point2 - (
              point1 - PT(0,0,0))));
630         if(cross.x() == 0 && cross.y() == 0 && cross.z() == 0)
631             result = true;
632     }
633     return result;
634 }
635
636
637
638 ///Input/output
639 template <class TRI>
640 bool WriteTriToFile(char* filename, char* writeMode, TRI* tris, int numTri, bool
        includeChecked)
641 {
642     FILE* output = fopen(filename, writeMode);
643     if(output != NULL)
644     {
```

```
645         if(writeMode[0] == 'a' && writeMode[1] == 't')
646             fprintf(output, "\t");
647 
648         for(int i = 0; i < numTri; i++)
649         {
650             if(&(tris[i]) >= RESERVED && (includeChecked || !tris[i]->checked) &&
                    tris[i]->CheckValidTri())
651             {
652                 if(tris[i]->GetPoint(0) == NULL || tris[i]->GetPoint(1) == NULL
                        || tris[i]->GetPoint(2) == NULL)
653                 {
654                     if(SHOW_MATH_ERRORS) printf("Error - A triangle exists
                            without three points\n");
655                 }
656                 else
657                 {
658                     fprintf(output, "%d %d %d\t", (int) tris[i]->GetPoint(0)->x()
                            , (int) tris[i]->GetPoint(0)->y(), (int) tris[i]->
                            GetPoint(0)->z());
659                     fprintf(output, "%d %d %d\t", (int) tris[i]->GetPoint(1)->x()
                            , (int) tris[i]->GetPoint(1)->y(), (int) tris[i]->
                            GetPoint(1)->z());
660                     fprintf(output, "%d %d %d\n", (int) tris[i]->GetPoint(2)->x()
                            , (int) tris[i]->GetPoint(2)->y(), (int) tris[i]->
                            GetPoint(2)->z());
661                 }
662             }
663         }
664         fclose(output);
665     }
666 
667     return true;
668 }
669 
670 template <class PT>
671 bool WritePtToFile(char* filename, char* writeMode, PT** points, int numPoints)
672 {
673     FILE* output = fopen(filename, writeMode);
674     if(output != NULL)
675     {
676         if(writeMode[0] == 'a' && writeMode[1] == 't')
677             fprintf(output, "\t");
678 
679         for(int i = 0; i < numPoints; i++)
680         {
681             if(points[i] >= RESERVED)
```

```
682                 fprintf(output, "%d %d %d\n", (int) points[i]->x(), (int) points[
                        i]->y(), (int) points[i]->z());
683
684             }
685             fclose(output);
686         }
687
688         return true;
689     }
690
691     bool WriteColourChangeToFile(char* filename);
692     bool WriteLineToFile(char* filename, char* writeMode, Edge** edges, int numEdges)
        ;
693     bool WriteLineToFile(char* filename, char* writeMode, Point** edges, int
        numEdgePairs);
694     void ScaleTriangles(int** tris, int numTriVertices, int width, int height, int
        xOffset, int yOffset);
695     void display(char* filename);
696     void recreateTraversedPath(char* filename);
697     void displayTopAndSide(char* filename);
698
699     ///Obsolete?
700     void displayYZSide(char* filename);
701
702     #endif
```

A.9.2 adv_math.cpp

```
1   /*************************************************************************
2    *   Copyright (C) 2007 by Michael Douglas Hasler   *
3    *   Redheadpunk@gmail.com   *
4    *                                                                       *
5    *   This program is free software; you can redistribute it and/or modify  *
6    *   it under the terms of the GNU General Public License as published by  *
7    *   the Free Software Foundation; either version 2 of the License, or     *
8    *   (at your option) any later version.                                   *
9    *                                                                       *
10   *   This program is distributed in the hope that it will be useful,       *
11   *   but WITHOUT ANy WARRANTy; without even the implied warranty of        *
12   *   MERCHANTABILITy or FITNESS FOR A PARTICULAR PURPOSE.  See the         *
13   *   GNU General Public License for more details.                          *
14   *                                                                       *
15   *   you should have received a copy of the GNU General Public License     *
16   *   along with this program; if not, write to the                         *
17   *   Free Software Foundation, Inc.,                                       *
18   *   59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.             *
```

```
19  ***********************************************************************/
20  #include "adv_math.h"
21
22      /** Recursively go through trying to get all triangles via linking between
            neighbours
23       */
24      void Triangle::RecursGetTri(Triangle*** allTriangles, int *numTri)
25      {
26          if(this > RESERVED && !checked)
27          {
28              if(*numTri != 0 && *numTri%50 == 0)
29              {
30                  void* tmp = realloc(*allTriangles, (*numTri+50) * sizeof(Triangle
                        *));
31                  if(tmp != NULL)
32                      *allTriangles = (Triangle**) tmp;
33                  else
34                      printf("Error - Could not reallocate memory for allTriangles\
                            n");
35              }
36
37              checked = true;
38              (*allTriangles)[(*numTri)++] = this;
39              neighbour1->RecursGetTri(allTriangles, numTri);
40              neighbour2->RecursGetTri(allTriangles, numTri);
41              neighbour3->RecursGetTri(allTriangles, numTri);
42          }
43
44          return;
45      }
46
47      Coord::Coord(int setX, int setY, int setZ)
48      {
49          xVal = setX;     yVal = setY;     zVal = setZ;
50      }
51
52      bool Coord::operator==(const Coord &other)
53      {
54          if(this->xVal == other.xVal && this->yVal == other.yVal && this->zVal ==
                other.zVal)
55              return true;
56          else
57              return false;
58      }
59
60      bool Coord::operator!=(const Coord &other)
61      {
```

```
62          return !(*this == other);
63      }
64
65      Coord Coord::operator+= (const Coord &other)
66      {
67          Coord C;
68          C.xVal = this->xVal + other.xVal;
69          C.yVal = this->yVal + other.yVal;
70          C.zVal = this->zVal + other.zVal;
71          return C;
72      }
73
74      Coord Coord::operator+ (const Coord &other)
75      {
76          Coord C;
77          C.xVal = this->xVal + other.xVal;
78          C.yVal = this->yVal + other.yVal;
79          C.zVal = this->zVal + other.zVal;
80          return C;
81      }
82
83      Coord Coord::operator- (const Coord &other)
84      {
85          Coord C;
86          C.xVal = this->xVal - other.xVal;
87          C.yVal = this->yVal - other.yVal;
88          C.zVal = this->zVal - other.zVal;
89          return C;
90      }
91
92      bool Coord::CheckVert(Coord point2, Coord point3)
93      {
94          bool result = false;
95          if(CheckAlongEdge(false, *this, point2, point3) || CheckAlongEdge(false,
                point2, *this, point3) || CheckAlongEdge(false, point3, *this, point2
                ))
96              result = true;
97          return result;
98      }
99
100     Coord Coord::CrossProduct(Coord otherCoord)
101     {
102         Coord C;
103         C.xVal =  yVal * otherCoord.zVal - zVal * otherCoord.yVal;
104         C.yVal = -xVal * otherCoord.zVal + zVal * otherCoord.xVal;
105         C.zVal =  xVal * otherCoord.yVal - yVal * otherCoord.xVal;
106         return C;
```

```
107     }
108
109     long Coord::DotProduct(Coord otherCoord)
110     {
111         long C = xVal * otherCoord.xVal + yVal * otherCoord.yVal + zVal *
                otherCoord.zVal;
112         return C;
113     }
114
115     bool Coord::CoordsVertOffset(Coord otherCoord)
116     {
117         if(xVal == otherCoord.xVal && yVal == otherCoord.yVal && zVal !=
                otherCoord.zVal)
118             return true;
119         else
120             return false;
121     }
122
123     Edge::Edge(Coord* ptA, Coord* ptB, bool left, bool right, Triangle* tri)
124     {
125         pointA = ptA;
126         pointB = ptB;
127         leftSideDone = left;
128         rightSideDone = right;
129         sideTri = tri;
130     }
131
132     Triangle::Triangle()
133     {
134         point1 = NULL;
135         point2 = NULL;
136         point3 = NULL;
137         neighbour1 = NULL;
138         neighbour2 = NULL;
139         neighbour3 = NULL;
140         hazardType = NONE;
141         checked = false;
142         vert = false;
143     }
144
145     Triangle::Triangle(Coord* pointA, Coord* pointB, Coord* pointC)
146     {
147         point1 = pointA;
148         point2 = pointB;
149         point3 = pointC;
150         neighbour1 = NULL;
151         neighbour2 = NULL;
```

```
152         neighbour3 = NULL;
153         hazardType = NONE;
154         checked = false;
155         vert = false;
156     }
157 
158     /** Set certain neighbour based on num given. Checking that neighbour isn't
            already set
159      * Should this return bool to let calling function know if had error or not?
160      */
161     void Triangle::SetNeighbour(Triangle* neigh, int num)
162     {
163         if(neighbor(num) == NULL || neighbor(num)->checked)
164         {
165             if(num == 0)
166                 neighbour1 = neigh;
167             else if(num == 1)
168                 neighbour2 = neigh;
169             else if(num == 2)
170                 neighbour3 = neigh;
171         }
172         else
173             if(SHOW_ITEM_ERRORS) printf("Error - Two neighbours found on same
                edge\n");
174     }
175 
176     /** Returns a neighbour, naming & syntax to make interchangeable with Faces
177      */
178     Triangle* Triangle::neighbor(int num)
179     {
180         Triangle* neighbour = NULL;
181         if(num == 0)
182             neighbour = neighbour1;
183         else if(num == 1)
184             neighbour = neighbour2;
185         else if(num == 2)
186             neighbour = neighbour3;
187 
188         return neighbour;
189     }
190 
191     Coord* Triangle::GetPoint(int num)
192     {
193         Coord* point = NULL;
194         if(num == 0)
195             point = point1;
196         else if(num == 1)
```

```
197             point = point2;
198         else if(num == 2)
199             point = point3;
200         return point;
201     }
202
203     void Triangle::DelinkTriangle()
204     {
205         if(neighbour1 != NULL)
206         {
207             if(neighbour1->neighbour1 > RESERVED && neighbour1->neighbour1->
                    CheckValidTri() && this->SameTri(neighbour1->neighbour1, true,
                    false))
208                 neighbour1->neighbour1 = NULL;
209             else if(neighbour1->neighbour2 > RESERVED && neighbour1->neighbour2->
                    CheckValidTri() && this->SameTri(neighbour1->neighbour2, true,
                    false))
210                 neighbour1->neighbour2 = NULL;
211             else if(neighbour1->neighbour3 > RESERVED && neighbour1->neighbour3->
                    CheckValidTri() && this->SameTri(neighbour1->neighbour3, true,
                    false))
212                 neighbour1->neighbour3 = NULL;
213             else if(!(neighbour1->neighbour1 == NULL && neighbour1->neighbour2 ==
                     NULL && neighbour1->neighbour3 == NULL))
214             {
215                 if(SHOW_MATH_WARNINGS) printf("Warning - A neighbour did not hold
                    a link back to current triangle\n");
216             }
217             neighbour1 = NULL;
218         }
219
220         if(neighbour2 != NULL)
221         {
222             if(neighbour2->neighbour1 > RESERVED  && neighbour2->neighbour1->
                    CheckValidTri() && this->SameTri(neighbour2->neighbour1, true,
                    false))
223                 neighbour2->neighbour1 = NULL;
224             else if(neighbour2->neighbour2 > RESERVED && neighbour2->neighbour2->
                    CheckValidTri() && this->SameTri(neighbour2->neighbour2, true,
                    false))
225                 neighbour2->neighbour2 = NULL;
226             else if(neighbour2->neighbour3 > RESERVED && neighbour2->neighbour3->
                    CheckValidTri() && this->SameTri(neighbour2->neighbour3, true,
                    false))
227                 neighbour2->neighbour3 = NULL;
228             else if(!(neighbour2->neighbour1 == NULL && neighbour2->neighbour2 ==
                     NULL && neighbour2->neighbour3 == NULL))
```

```
229             {
230                 if(SHOW_MATH_WARNINGS) printf("Warning - A neighbour did not hold
                        a link back to current triangle\n");
231             }
232             neighbour2 = NULL;
233         }
234 
235         if(neighbour3 != NULL)
236         {
237             if(neighbour3->neighbour1 > RESERVED && neighbour3->neighbour1->
                    CheckValidTri() && this->SameTri(neighbour3->neighbour1, true,
                    false))
238                 neighbour3->neighbour1 = NULL;
239             else if(neighbour3->neighbour2 > RESERVED && neighbour3->neighbour2->
                    CheckValidTri() && this->SameTri(neighbour3->neighbour2, true,
                    false))
240                 neighbour3->neighbour2 = NULL;
241             else if(neighbour3->neighbour3 > RESERVED && neighbour3->neighbour3->
                    CheckValidTri() && this->SameTri(neighbour3->neighbour3, true,
                    false))
242                 neighbour3->neighbour3 = NULL;
243             else if(!(neighbour3->neighbour1 == NULL && neighbour3->neighbour2 ==
                     NULL && neighbour3->neighbour3 == NULL))
244             {
245                 if(SHOW_MATH_WARNINGS) printf("Warning - A neighbour did not hold
                        a link back to current triangle\n");
246             }
247             neighbour3 = NULL;
248         }
249         checked = true;
250         point1 = NULL;
251         point2 = NULL;
252         point3 = NULL;
253     }
254 
255     /** Compare a triangle to see if it has two matching points and thus is a
            neighbouring triangle
256      */
257     bool Triangle::CheckIfNeighbours(Triangle* test)
258     {
259         bool result = false;
260         int matches = 0;
261         if(test != NULL)
262         {
263             if(point1 == test->point1 || point1 == test->point2 || point1 == test
                    ->point3)
264                 matches++;
```

```
265             if(point2 == test->point1 || point2 == test->point2 || point2 == test
                    ->point3)
266                 matches++;
267             if(point3 == test->point1 || point3 == test->point2 || point3 == test
                    ->point3)
268                 matches++;
269             if(matches == 2)
270             {
271                 result = true;
272             }
273         }
274         return result;
275     }
276
277     bool Triangle::CheckValidTri()
278     {
279         bool result = true;
280         if(this == NULL || !(point1 > RESERVED && point2 > RESERVED && point3 >
                RESERVED) || !((neighbour1 == NULL || neighbour1 > RESERVED) && (
                neighbour2 == NULL || neighbour2 > RESERVED) && (neighbour3 == NULL
                || neighbour3  > RESERVED)) || hazardType >= MAX_HAZARD  ||
                hazardType < MIN_HAZARD)
281             result = false;
282         return result;
283     }
284
285     bool Triangle::CheckVert()
286     {
287         bool result = false;
288         if(CheckAlongEdge(false, *point1, *point2, *point3) || CheckAlongEdge(
                false, *point2, *point1, *point3) || CheckAlongEdge(false, *point3, *
                point1, *point2))
289             result = true;
290         return result;
291     }
292
293     /** Check a triangles neighbours to see they are infact neighbouring ie share
                an edge aka two points
294      */
295     bool Triangle::TestNeighboursCorrect()
296     {
297         bool result = true;
298         if(neighbour1 != NULL && !CheckIfNeighbours(neighbour1))
299         {
300                 if(SHOW_MATH_ERRORS) printf("Error - Triangle and neighbour1 are
                        wrongly linked\n");
301                 result = false;
```

```
302         }
303
304         if(neighbour2 != NULL && !CheckIfNeighbours(neighbour2))
305         {
306                 if(SHOW_MATH_ERRORS) printf("Error - Triangle and neighbour2 are
                        wrongly linked\n");
307                 result = false;
308         }
309
310         if(neighbour3 != NULL && !CheckIfNeighbours(neighbour3))
311         {
312                 if(SHOW_MATH_ERRORS) printf("Error - Triangle and neighbour3 are
                        wrongly linked\n");
313                 result = false;
314         }
315
316         return result;
317     }
318
319     /** Checks which points are common between two triangles.
320      * PtsA holding which of the otherTri are common to it,and PtsB is vice versa
321      * notCommon holds the points of the first triangle (this) which are not in
             common, if only one number is held,
322      * then that shows which neighbour otherTri should be stored as
323      */
324     void Triangle::CommonPoints(Triangle* otherTri, int* ptsA, int* ptsB, int*
            notCommonA, int* notCommonB)
325     {
326         int numPts = 0;
327         int numNotCommA = 0;
328         int numNotCommB = 0;
329
330         for(int i = 0; i < 3; i++)
331         {
332             int j;
333             for(j = 0; j < 3; j++)
334             {
335                 if(GetPoint(i) == otherTri->GetPoint(j))
336                 {
337                     ptsA[numPts] = i;
338                     ptsB[numPts++] = j;
339                     break;
340                 }
341             }
342             if(j == 3)
343                 notCommonA[numNotCommA++] = i;
344         }
```

```
345
346         if(numNotCommA > 0)
347         {
348             for(int i = 0; i < 3 && numNotCommB < numNotCommA; i++)
349             {
350                 if(ptsB[0] != i && ptsB[1] != i && ptsB[2] != i)
351                     notCommonB[numNotCommB++] = i;
352             }
353         }
354     }
355
356     void Triangle::LinkTwoTriangles(Triangle* otherTri)
357     {
358         int commonA[3] = {-1,-1,-1};
359         int commonB[3] = {-1,-1,-1};
360         int unCommonA[3] = {-1,-1,-1};
361         int unCommonB[3] = {-1,-1,-1};
362         this->CommonPoints(otherTri, commonA, commonB, unCommonA, unCommonB);
363
364         if(unCommonA[0] > -1 && unCommonA[1] == -1)
365         {
366             SetNeighbour(otherTri, unCommonA[0]);
367             SetNeighbour(otherTri, unCommonB[0]);
368         }
369         else
370         {
371             if(SHOW_MATH_ERRORS) printf("Error - Triangles are not neighbours,
                    either 0,1 or 3 pts in common\n");
372         }
373     }
374
375     /** Compare two triangles, the first of which is the held/kept one and the
            second is one being considered for keeping
376         Passed flags of whether rearranged coords should be considered matches or
                only identical ones
377         If non-identical ones are being compared, the first one may be changed to
                have the neighbours of second added to it.
378      */
379     bool Triangle::SameTri(Triangle* triToTest, bool identicalMatchesOnly, bool
            modDiffInstanceSameTri)
380     {
381         bool result = false;
382         if(this == triToTest)
383             result = true;
384         else if(point1 > RESERVED && point2 > RESERVED && point3 > RESERVED &&
                triToTest > RESERVED && triToTest->point1 > RESERVED && triToTest->
                point2 > RESERVED && triToTest->point3 > RESERVED)
```

```
385         {
386             if(*point1 == *triToTest->point1 && *point2 == *triToTest->point2 &&
                    *point3 == *triToTest->point3)
387             {
388                 SameTriNeighboursCheck(triToTest);
389                 result = true;
390             }
391             else if(!identicalMatchesOnly && ((*point1 == *triToTest->point1 || *
                    point1 == *triToTest->point2 || *point1 == *triToTest->point3) &&
                     (*point2 == *triToTest->point1 || *point2 == *triToTest->point2
                    || *point2 == *triToTest->point3) && (*point3 == *triToTest->
                    point1 || *point3 == *triToTest->point2 || *point3 == *triToTest
                    ->point3)))
392             {
393                 if(SHOW_MATH_WARNINGS) printf("Warning - Two identical triangles,
                        just with rearranged coords, were found\n");
394                 if(modDiffInstanceSameTri)
395                     SameTriNeighboursCheck(triToTest);
396                 result = true;
397             }
398             else
399                 result = false;
400         }
401         else
402             if(SHOW_MATH_ERRORS) printf("Error - A triangle being compared had
                    NULL points or INVALID points\n");
403 
404         return result;
405     }
406 
407     /** Check if neighbours point to same tris, are same tris or if not, add
            unheld tri and re-link said tri to point to tri A.
408         Will there be problems of recursive calls or unneeded processing, which
                could be avoided by not doing SametriNeighbour check on neighbours of
                 neighbours etc?
409      */
410     bool Triangle::SameTriNeighboursCheck(Triangle* triToTest)
411     {
412         bool result = false;
413         if(point1 > RESERVED && point2 > RESERVED && point3 > RESERVED &&
                triToTest->point1 > RESERVED && triToTest->point2 > RESERVED &&
                triToTest->point3 > RESERVED)
414         {
415             int commonA[3] = {-1,-1,-1}, commonB[3] = {-1,-1,-1};
416             int unCommonA[3] = {-1,-1,-1}, unCommonB[3] = {-1,-1,-1};
417             triToTest->CommonPoints(this, commonA, commonB, unCommonA, unCommonB)
                    ;
```

```
418             if(unCommonA[0] > -1 && unCommonA[1] == -1 &&
                    SameTriSameNeighbourCheck(triToTest, triToTest->neighbor(
                    unCommonA[0])))
419                 result = true;
420         }
421
422         return result;
423     }
424
425     /** Check if neighbours point to same tris, are same tris or if not, add
            unheld tri.
426         To be true, triNeighbourToTest must be the same as calling tri(this) and
                triToTest and this must be linked accordingly
427         Will there be problems of recursive calls or unneeded processing, which
                could be avoided buy not doing SametriNeighbour check on neighbours
                of neighbours etc?
428      */
429     bool Triangle::SameTriSameNeighbourCheck(Triangle* triToTest, Triangle*
            triNeighbourToTest)
430     {
431         bool result = true;
432         if(!(triNeighbourToTest <= RESERVED || triNeighbourToTest->checked) && !
                checked)
433         {
434             int commonA[3] = {-1,-1,-1}, commonB[3] = {-1,-1,-1};
435             int unCommonA[3] = {-1,-1,-1}, unCommonB[3] = {-1,-1,-1};
436             this->CommonPoints(triNeighbourToTest, commonA, commonB, unCommonA,
                    unCommonB);
437             if(unCommonA[0] > -1 && unCommonA[1] == -1)
438             {
439                 if( ! (triNeighbourToTest == neighbor(unCommonA[0]) || (neighbor(
                        unCommonA[0]) > RESERVED && !neighbor(unCommonA[0])->checked
                        && triNeighbourToTest->SameTri(neighbor(unCommonA[0]), false,
                         true) && triNeighbourToTest->SameTriNeighboursCheck(neighbor
                        (unCommonA[0])))) )
440                 {
441                     SetNeighbour(triNeighbourToTest, unCommonA[0]);
442
443                     if(triNeighbourToTest->neighbour1 != triToTest &&
                            triNeighbourToTest->neighbour2 != triToTest &&
                            triNeighbourToTest->neighbour3 != triToTest)
444                         if(SHOW_MATH_ERRORS) printf("Error - triNeighbourToTest
                                did not point back to triToTest as a neighbour\n");
445
446                     triNeighbourToTest->SetNeighbour(this, unCommonB[0]);
447                 }
448             }
```

```
449         }
450         else
451             result = false;
452 
453         return result;
454     }
455 
456     int Triangle::InterpolateZValue(Coord retrievalPoint)
457     {
458         Coord norm, v0, v1, v2;
459         v0 = *point2 - *point1;
460         v1 = *point3 - *point1;
461         norm = v0.CrossProduct(v1);
462         retrievalPoint.zVal = 0;//otherwise zValue is difference between tri and
                retrieval pt
463         v2 = *point1 - retrievalPoint;
464         int dot;
465         dot = norm.DotProduct(v2);
466         int ZValue = dot/norm.zVal;
467 
468         return ZValue;
469     }
470 
471     /** Given an array of triangles, match neighbours and link them
472      * Triangles which were part of cavity edge, will have some non-NULL
             neighbours stored already whereas newly created Tri will not
473      * Act accordingly
474      */
475     Triangle** LinkTriangles(Triangle** newTris, int triIndex, int numEdge)
476     {
477         for(int i = 0; i < triIndex; i++)
478         {
479             if(!newTris[i]->checked)
480             {
481                 if(newTris[i]->point1 == NULL || newTris[i]->point2 == NULL ||
                        newTris[i]->point3 == NULL)
482                 {
483                     if(SHOW_MATH_WARNINGS) printf("Warning - A triangle exists
                            without three points\n");
484                 }
485                 else
486                 {
487                     for(int j = i+1; j < triIndex; j++)
488                     {
489                         if(!newTris[j]->checked)
490                         {
491                             int commonA[3] = {-1,-1,-1}, commonB[3] = {-1,-1,-1};
```

```
492                             int unCommonA[3] = {-1,-1,-1}, unCommonB[3] =
                                    {-1,-1,-1};
493                             newTris[i]->CommonPoints(newTris[j], commonA, commonB
                                    , unCommonA, unCommonB);
494                             if(unCommonA[0] > -1 && unCommonA[1] == -1)
495                             {
496                                 newTris[i]->SetNeighbour(newTris[j], unCommonA
                                        [0]);
497                                 newTris[j]->SetNeighbour(newTris[i], unCommonB
                                        [0]);
498                             }
499                         }
500                     }
501                 }
502             }
503         }
504
505         for(int i = 0; i < triIndex; i++)
506         {
507             if(newTris[i] != NULL && !newTris[i]->checked)
508             {
509                 if(newTris[i]->vert)
510                 {
511                     if(!((newTris[i]->neighbour1 != NULL && !newTris[i]->
                            neighbour1->vert && !newTris[i]->neighbour1->checked) ||
                            (newTris[i]->neighbour2 != NULL && !newTris[i]->
                            neighbour2->vert && !newTris[i]->neighbour2->checked) ||
                            (newTris[i]->neighbour3 != NULL && !newTris[i]->
                            neighbour3->vert && !newTris[i]->neighbour3->checked)))
512                     {
513                         if(newTris[i]->neighbour1 != NULL && !newTris[i]->
                                neighbour1->checked && newTris[i]->neighbour2 != NULL
                                 && !newTris[i]->neighbour2->checked && newTris[i]->
                                neighbour3 != NULL && !newTris[i]->neighbour3->
                                checked)
514                         {
515                             if(SHOW_MATH_WARNINGS) printf("Warning - A vertical
                                    triangles has three neighbours which are all
                                    vertical\n");
516                         }
517                         else
518                             newTris[i]->checked = true;
519                     }
520                 }
521                 else if((newTris[i]->neighbour1 == NULL || newTris[i]->neighbour1
                        ->checked) && (newTris[i]->neighbour2 == NULL || newTris[i]->
                        neighbour2->checked) && (newTris[i]->neighbour3 == NULL ||
```

```
                    newTris[i]->neighbour3->checked))
522                 if(SHOW_MATH_ERRORS) printf("Error - After linking, one non-
                        Vert triangle in newTris did not have neighbours\n");
523             }
524         }
525
526         WriteTriToFile("triangles2.txt", "wt", newTris, triIndex, false);
527
528         return newTris;
529     }
530
531     Triangle** SortTriangles(Triangle** newTris, int triIndex)
532     {
533         Triangle* tempPtr;
534
535         for(int i = 0, j = triIndex - 1; i < triIndex && j > i; i++)
536         {
537             if(!newTris[i]->checked &&  newTris[i]->vert)//CheckVert(newTris[i]))
538             {
539                 if(newTris[j]->checked ||  !newTris[j]->vert)//!CheckVert(newTris
                        [j]))
540                 {
541                     tempPtr = newTris[i];
542                     newTris[i] = newTris[j];
543                     newTris[j] = tempPtr;
544                 }
545
546                 j--;
547             }
548             else
549                 i++;
550         }
551
552         for(int i = 0; i < triIndex; i++)
553         {
554             if(newTris[i] != NULL && ((newTris[i]->neighbour1 != NULL && (newTris
                    [i]->neighbour1 == newTris[i]->neighbour2 || newTris[i]->
                    neighbour1 == newTris[i]->neighbour3)) || (newTris[i]->neighbour2
                     != NULL && newTris[i]->neighbour2 == newTris[i]->neighbour3)))
555                 if(SHOW_MATH_ERRORS) printf("Error - Two of same triangle in
                        newTris after linking\n");
556         }
557
558         return newTris;
559     }
560
```

```
561     /** Tests four not nine combos of edges, as either two edges or no edges will
            cross as no point can be inside the trianges
562      */
563     int CheckTriangleEdgeIntersections(bool samePlane, Coord tri1_A, Coord tri1_B
            , Coord tri1_C, Coord tri2_A, Coord tri2_B, Coord tri2_C)
564     {
565         //check if one tri is vert translation of the other
566
567         int result = 0;
568         int lower = 0;
569         if(CheckEdgeIntersections(samePlane, false, tri1_A, tri1_B, tri2_A,
                tri2_B, &lower) || CheckEdgeIntersections(samePlane, false, tri1_A,
                tri1_B, tri2_A, tri2_C, &lower) || CheckEdgeIntersections(samePlane,
                false, tri1_A, tri1_C, tri2_A, tri2_B, &lower) ||
                CheckEdgeIntersections(samePlane, false, tri1_A, tri1_C, tri2_A,
                tri2_C, &lower))
570             result = lower;
571
572         return result;
573     }
574
575     /** Tri1 being the older, known tri and Tri2 being the tested one**/
576     int CalcBetterTriangle(Coord tri1_A, Coord tri1_B, Coord tri1_C, Coord tri2_A
            , Coord tri2_B, Coord tri2_C)
577     {
578         int result = 1;
579         float temp[2] = {-1,-1};
580         CalculateDistsToCentre(tri1_A, tri1_B, tri1_C, temp);
581         float dist1 = temp[0];
582         float dist1_min = temp[1];
583         CalculateDistsToCentre(tri2_A, tri2_B, tri2_C, temp);
584         float dist2 = temp[0];
585         float dist2_min = temp[1];
586
587         if(dist2/dist2_min > 5)
588             result = 11;
589         else if(dist1/dist1_min > 5)
590             result = 12;
591         else
592         {
593             if(dist2 >= dist1)
594                 result = 1;
595             else
596             {
597                 double area1 = abs((tri1_B.xVal*tri1_A.yVal-tri1_A.xVal*tri1_B.
                        yVal)+(tri1_C.xVal*tri1_B.yVal-tri1_B.xVal*tri1_C.yVal)+(
                        tri1_A.xVal*tri1_C.yVal-tri1_C.xVal*tri1_A.yVal))/2;
```

```
598                 double area2 = abs((tri2_B.xVal*tri2_A.yVal-tri2_A.xVal*tri2_B.
                        yVal)+(tri2_C.xVal*tri2_B.yVal-tri2_B.xVal*tri2_C.yVal)+(
                        tri2_A.xVal*tri2_C.yVal-tri2_C.xVal*tri2_A.yVal))/2;
599                 if(area1/dist1 > area2/dist2)
600                     result = 2;
601                 else
602                     result = 1;
603             }
604         }
605         return result;
606     }
607
608     bool Edge::SameEdge(Edge* newEdge, bool modEdges)
609     {
610         bool result = false;
611         if(*pointA == *newEdge->pointA && *pointB  == *newEdge ->pointB)
612         {
613             result = true;
614             if(modEdges)
615             {
616                 if(leftSideDone != rightSideDone && leftSideDone == newEdge->
                        rightSideDone && rightSideDone == newEdge->leftSideDone)//
                        they each meet edge from different side
617                 {
618                     leftSideDone = true;
619                     rightSideDone = true;
620                     newEdge->leftSideDone = true;
621                     newEdge->rightSideDone = true;
622                     sideTri->LinkTwoTriangles(newEdge->sideTri);
623                 }
624                 else
625                     if(SHOW_MATH_ERRORS) printf("Error - Matching edges were
                            found, however they didn't match up in terms of which
                            sides had triangles formed on them\n");
626             }
627         }
628         else if(*pointA  == *newEdge->pointB && *pointB == *newEdge->pointA)
629         {
630             result = true;
631             if(modEdges)
632             {
633                 //compare which sides have been done, though they are reversed
                        due to order of points in edge
634                 if(leftSideDone != rightSideDone && leftSideDone == newEdge->
                        leftSideDone && rightSideDone == newEdge->rightSideDone)//
                        they each meet edge from different side
635                 {
```

```
636                     leftSideDone = true;
637                     rightSideDone = true;
638                     newEdge->leftSideDone = true;
639                     newEdge->rightSideDone = true;
640                     sideTri->LinkTwoTriangles(newEdge->sideTri);
641                 }
642                 else
643                     if(SHOW_MATH_ERRORS) printf("Error - Matching edges were
                        found, however they didn't match up in terms of which
                        sides had triangles formed on them\n");
644             }
645         }
646
647         return result;
648     }
649
650     Coord FillCoord(int xVal, int yVal, int zVal, int scale)
651     {
652         Coord fill;
653         fill.xVal = xVal * scale;
654         fill.yVal = yVal * scale;
655         fill.zVal = zVal * scale;
656         return fill;
657     }
658
659 /// ******************** Input/Output ************************
660     bool WriteColourChangeToFile(char* filename)
661     {
662         FILE* output = fopen(filename, "at");
663         if(output != NULL)
664         {
665             fprintf(output, "%d\n", -1);
666             fclose(output);
667         }
668
669         return true;
670     }
671
672     bool WriteLineToFile(char* filename, char* writeMode, Edge** edges, int
        numEdge)
673     {
674         FILE* output = fopen(filename, writeMode);
675         if(output != NULL)
676         {
677             if(writeMode[0] == 'a' && writeMode[1] == 't')
678                 fprintf(output, "\t");
679
```

```
680             for(int i = 0; i < numEdge; i++)
681             {
682                 if(edges[i] >= RESERVED)
683                 {
684                     if(edges[i]->pointA == NULL || edges[i]->pointB == NULL)
685                     {
686                         if(SHOW_MATH_ERRORS) printf("Error - An edge exists
                            without two points\n");
687                     }
688                     else
689                     {
690                         fprintf(output, "%d %d %d\t", edges[i]->pointA->x(),
                            edges[i]->pointA->y(), edges[i]->pointA->z());
691                         fprintf(output, "%d %d %d\n", edges[i]->pointB->x(),
                            edges[i]->pointB->y(), edges[i]->pointB->z());
692                     }
693                 }
694             }
695             fclose(output);
696         }
697
698         return true;
699     }
700
701     bool WriteLineToFile(char* filename, char* writeMode, Point** edges, int
        numEdgePoints)
702     {
703         FILE* output = fopen(filename, writeMode);
704         if(output != NULL)
705         {
706             if(writeMode[0] == 'a' && writeMode[1] == 't')
707                 fprintf(output, "\t");
708
709             for(int i = 0; i < numEdgePoints; i += 2)
710             {
711                 if(edges[i] >= RESERVED)
712                 {
713                     if(edges[i] == NULL || edges[i+1] == NULL)
714                     {
715                         if(SHOW_MATH_ERRORS) printf("Error - An edge exists
                            without two points\n");
716                     }
717                     else
718                     {
719                         int x = edges[i]->x(), y = edges[i]->y(), z = edges[i]->z
                            ();
720                         fprintf(output, "%d %d %d\t", x,y,z); i++;
```

```
721                         x = edges[i]->x(), y = edges[i]->y(), z = edges[i]->z();
722                         fprintf(output, "%d %d %d\n", x,y,z);
723                     }
724                 }
725             }
726             fclose(output);
727         }
728 
729         return true;
730     }
731 
732     void ScaleTriangles(int** tris, int numTriVertices, int width, int height,
        int xOffset, int yOffset)
733     {
734         float xmin = -1, xmax = -1, ymin = -1, ymax = -1;
735         for(int j = 0; j < numTriVertices*2; j += 2)
736         {
737             if(((*tris)[j] < xmin || xmin == -1) && (*tris)[j] != 0)
738                 xmin = (*tris)[j];
739             if((*tris)[j] > xmax)
740                 xmax = (*tris)[j];
741             if(((*tris)[j+1] < ymin || ymin == -1) && (*tris)[j+1] != 0)
742                 ymin = (*tris)[j+1];
743             if((*tris)[j+1] > ymax)
744                 ymax = (*tris)[j+1];
745         }
746 
747         if(xmin < 0 || ymin < 0 || xmax < 0 || ymax < 0)
748         {
749             if(SHOW_MATH_ERRORS) printf("Error - One of min/max limits for coords
                was not set\n");
750         }
751         else if(xmin == xmax || ymin == ymax)
752         {
753             if(SHOW_MATH_ERRORS) printf("Error - The min & max limits for either
                X or Y were the same. Either a vertical triangle or line was in '
                tris', or soem of the coords were zero\n");
754         }
755         else
756         {
757             float fitWidth = width/(xmax - xmin);
758             float fitHeight = height/(ymax - ymin);
759 
760             for(int j = 0; j < numTriVertices*2; j += 2)
761             {
762                 if((*tris)[j] != 0)
763                 {
```

```
764                     (*tris)[j] -= xmin;
765                     (*tris)[j] = (*tris)[j]*fitWidth + xOffset;
766                 }
767
768                 if((*tris)[j+1] != 0)
769                 {
770                     (*tris)[j+1] -= ymin;
771                     (*tris)[j+1] = (*tris)[j+1]*fitHeight + yOffset;
772                 }
773             }
774         }
775     }
776
777     void display(char* filename)
778     {
779         int* tris = (int*) calloc(6 * 10, sizeof(int));
780         int numCoords = 0;
781         int colourChange[255];
782         int numChanges = 0;
783         FILE* triangles = fopen(filename, "rt");
784         if(triangles == NULL)
785         {
786             if(SHOW_MATH_ERRORS) printf("Error - Unable to open triangles2.txt\n"
                    );
787             //exit(1); // terminate with error
788         }
789         else
790         {
791             int i = 0;
792             MODES drawShape = TRI_M;
793             char buffer[90];
794             while(fgets(buffer, 90, triangles) != NULL)
795             {
796                 if(buffer[0] != '\n')
797                 {
798                     if(i%60 == 0 && i != 0)
799                     {
800                         void* tmp = realloc(tris, (i + 6 * 10) * sizeof(int));
801                         if(tmp != NULL)
802                             tris = (int*) tmp;
803                         else
804                             if(SHOW_MATH_ERRORS) printf("Error - Unable to
                                    reallocate memory for triangle coords\n");
805                     }
806
807                     int numScanPts = sscanf(buffer, "%d %d %*d %d %d %*d %d %d %*
                        d", &tris[i], &tris[i+1], &tris[i+2], &tris[i+3], &tris[i
```

```
                        +4], &tris[i+5]);
808
809                     if(numScanPts == 6)
810                     {
811                         i += 6;
812                         numCoords += 3;
813                     }
814                     else if(numScanPts == 4)
815                     {
816                         drawShape = LINE_M;
817                         i += 4;
818                         numCoords += 2;
819                     }
820                     else if(numScanPts == 2)
821                     {
822                         drawShape = POINT_M;
823                         i += 2;
824                         numCoords += 1;
825                     }
826                     else if(numScanPts == 1)
827                         colourChange[numChanges++] = numCoords;
828                 }
829             }
830             fclose(triangles);
831
832             ScaleTriangles(&tris, numCoords, 700, 700, 50, 50);
833             runDisplay(tris, numCoords, drawShape, colourChange, numChanges);
834         }
835
836         free(tris);
837         tris = NULL;
838     }
839
840     void recreateTraversedPath(char* filename)
841     {
842         int chunk = 4 * 10;
843         int* tris = (int*) calloc(chunk, sizeof(int));
844         int numCoords = 0;
845         int colourChange[255];
846         int numChanges = 0;
847         FILE* triangles = fopen(filename, "rt");
848         if(triangles == NULL)
849         {
850             if(SHOW_MATH_ERRORS) printf("Error - Unable to open file containing
                    steps of traversed path.txt\n");
851         }
852         else
```

```
853         {
854             int i = 0;
855             char buffer[90];
856             while(fgets(buffer, 90, triangles) != NULL)
857             {
858                 if(buffer[0] != '\n')
859                 {
860                     int x = 0, y = 0;
861                     int numScanPts = sscanf(buffer, "%*s %d %d %*d", &x, &y);
862                     if(numScanPts == 2)
863                     {
864                         if(x > 10 || y > 10)
865                         {
866                             tris[i++] = x/10;
867                             tris[i++] = y/10;
868                         }
869                         else
870                         {
871                             tris[i++] = tris[i-2] + x;
872                             tris[i++] = tris[i-2] + y;
873                         }
874 
875                         if(i%chunk == 0)
876                         {
877                             void* tmp = realloc(tris, (i + chunk) * sizeof(int));
878                             if(tmp != NULL)
879                                 tris = (int*) tmp;
880                             else
881                                 if(SHOW_MATH_ERRORS) printf("Error - Unable to
                                        reallocate memory for triangle coords\n");
882                         }
883 
884                         if(i > 2)
885                         {
886                             tris[i++] = tris[i-2];
887                             tris[i++] = tris[i-2];
888                             numCoords += 1;
889                         }
890                     }
891                     else if(numScanPts == 0 && buffer[0] == 'T')
892                         colourChange[numChanges++] = numCoords;
893                 }
894             }
895 
896             fclose(triangles);
897 
```

```
898             ScaleTriangles(&tris, numCoords, 700, 700, 50, 50);//scaled a lot of
                    values to zero
899             runDisplay(tris, numCoords, LINE_M, colourChange, numChanges);//make
                    handle multiple colour changes?
900         }
901
902         free(tris);
903         tris = NULL;
904     }
905
906     void displayTopAndSide(char* filename)
907     {
908         int* tris = (int*) calloc(6 * 10, sizeof(int));
909         int* tris2 = (int*) calloc(6 * 10, sizeof(int));
910         int* tris3 = (int*) calloc(6 * 10, sizeof(int));
911         int* tris4 = (int*) calloc(6 * 10, sizeof(int));
912         int numTri = 0;
913         FILE* triangles = fopen(filename, "rt");
914         if(triangles == NULL)
915         {
916             if(SHOW_MATH_ERRORS) printf("Error - Unable to open triangles2.txt\n"
                    );
917             exit(1); // terminate with error
918         }
919
920         int i = 0;
921         char buffer[90];
922         bool scanFine = true;
923         fgets(buffer, 90, triangles);//get rid of first line
924         while(fgets(buffer, 90, triangles) != NULL)
925         {
926             if(buffer[0] != '\n')
927             {
928                 if(i%60 == 0 && i != 0)
929                 {
930                     void* tmp = realloc(tris, (i + 6 * 10) * sizeof(int));
931                     void* tmp2 = realloc(tris2, (i + 6 * 10) * sizeof(int));
932                     void* tmp3 = realloc(tris3, (i + 6 * 10) * sizeof(int));
933                     void* tmp4 = realloc(tris4, (i + 6 * 10) * sizeof(int));
934                     if(tmp != NULL && tmp2 != NULL && tmp3 != NULL && tmp4 !=
                            NULL)
935                     {
936                         tris = (int*) tmp;
937                         tris2 = (int*) tmp2;
938                         tris3 = (int*) tmp3;
939                         tris4 = (int*) tmp4;
940                     }
```

```
941                     else
942                         if(SHOW_MATH_ERRORS) printf("Error - Unable to reallocate
                                memory for triangle coords\n");
943                 }
944
945                 if(sscanf(buffer, "%d %d %*d %d %d %*d %d %d %*d", &tris[i], &
                        tris[i+1], &tris[i+2], &tris[i+3], &tris[i+4], &tris[i+5]) ==
                         6)
946                 {
947                     tris[i] += 200;
948                     tris[i+2] += 200;
949                     tris[i+4] += 200;
950                 }
951                 else
952                     scanFine = false;
953
954                 if(sscanf(buffer, "%*d %d %d %*d %d %d %*d %d %d", &tris2[i+1], &
                        tris2[i], &tris2[i+3], &tris2[i+2], &tris2[i+5], &tris2[i+4])
                         == 6)
955                 {
956                     tris3[i] = tris2[i] + 600;
957                     tris3[i+1] = tris2[i+1];
958                     tris3[i+2] = tris2[i+2] + 600;
959                     tris3[i+3] = tris2[i+3];
960                     tris3[i+4] = tris2[i+4] + 600;
961                     tris3[i+5] = tris2[i+5];
962                     tris2[i] = 200 - tris2[i];
963                     tris2[i+2] = 200 - tris2[i+2];
964                     tris2[i+4] = 200 - tris2[i+4];
965                 }
966                 else
967                     scanFine = false;
968
969                 if(sscanf(buffer, "%d %*d %d %d %*d %d %d %*d %d", &tris4[i], &
                        tris4[i+1], &tris4[i+2], &tris4[i+3], &tris4[i+4], &tris4[i
                        +5]) == 6)
970                 {
971                     tris4[i] += 200;
972                     tris4[i+1] += 520;
973                     tris4[i+2] += 200;
974                     tris4[i+3] += 520;
975                     tris4[i+4] += 200;
976                     tris4[i+5] += 520;
977                 }
978                 else
979                     scanFine = false;
980
```

```
981                 if(scanFine)
982                 {
983                     numTri += 12; i += 6;
984                 }
985                 else
986                     if(SHOW_MATH_ERRORS) printf("Error - Problem scanning in
                        coordinates from file\n");
987             }
988         }
989         fclose(triangles);
990 
991         int numTriVerts = i/2;
992         ScaleTriangles(&tris, numTriVerts, 400, 400, 200, 0);
993         ScaleTriangles(&tris2, numTriVerts, 200, 400, 0, 0);
994         ScaleTriangles(&tris3, numTriVerts, 200, 400, 600, 0);
995         ScaleTriangles(&tris4, numTriVerts, 400, 200, 200, 400);
996 
997         void* tmp = realloc(tris, (i*4) * sizeof(int));
998         if(tmp != NULL)
999             tris = (int*) tmp;
1000        else
1001            if(SHOW_MATH_ERRORS) printf("Error - Unable to reallocate memory for
                    triangle coords\n");
1002
1003        for(int j = i, k = 0; k < i; j++, k++)
1004        {
1005            tris[j] = tris2[k];
1006            tris[j + i] = tris3[k];
1007            tris[j + i*2] = tris4[k];
1008        }
1009
1010        runDisplay(tris, numTri, TRI_M, &i, 1);
1011        free(tris);
1012        free(tris2);
1013        free(tris3);
1014        free(tris4);
1015        tris = NULL;
1016        tris2 = NULL;
1017        tris3 = NULL;
1018        tris4 = NULL;
1019    }
1020
1021    void displayYZSide(char* filename)
1022    {
1023        int* tris = (int*) calloc(6 * 10, sizeof(int));
1024        int numTri = 0;
1025        int colourChange[255];
```

```
1026        int numChanges = 0;
1027        FILE* triangles = fopen(filename, "rt");
1028        if(triangles == NULL)
1029        {
1030            if(SHOW_MATH_ERRORS) printf("Error - Unable to open triangles2.txt\n"
                );
1031            //exit(1); // terminate with error
1032        }
1033        else
1034        {
1035            int i = 0;
1036            char buffer[90];
1037            while(fgets(buffer, 90, triangles) != NULL)
1038            {
1039                if(buffer[0] != '\n')
1040                {
1041                    if(i%60 == 0 && i != 0)
1042                    {
1043                        void* tmp = realloc(tris, (i + 6 * 10) * sizeof(int));
1044                        if(tmp != NULL)
1045                            tris = (int*) tmp;
1046                        else
1047                            if(SHOW_MATH_ERRORS) printf("Error - Unable to
                                reallocate memory for triangle coords\n");
1048                    }
1049
1050                    if(sscanf(buffer, "%*d %d %d %*d %d %d %*d %d %d", &tris[i
                        +1], &tris[i], &tris[i+3], &tris[i+2], &tris[i+5], &tris[
                        i+4]) == 6)
1051                    {
1052                        tris[i] += 400;
1053                        tris[i+2] += 400;
1054                        tris[i+4] += 400;
1055                        numTri += 3;
1056                        i += 6;
1057                    }
1058                    else
1059                        colourChange[numChanges++] = numTri;
1060                }
1061            }
1062            fclose(triangles);
1063            runDisplay(tris, numTri, TRI_M, colourChange, numChanges);
1064        }
1065        free(tris);
1066        tris = NULL;
1067    }
```

Appendix B

Simulation Results

B.1 Statistical Data

The statistical data and graphs are provided in spreadsheet form in the Results file of either .xls or .ods extension.